VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

PŘEKLADAČ Z OCTAVE DO C++

OCTAVE TO C++ SOURCE-TO-SOURCE COMPILER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **VÁCLAV ŠEVČÍK**
AUTHOR

VEDOUCÍ PRÁCE **MARTIN KOLÁŘ, M.Sc.**
SUPERVISOR

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Ševčík Václav**
Obor: Informační technologie
Téma: **Překladač z Octave do C++**
Octave to C++ Source-to-Source Compiler
Kategorie: Překladače

Pokyny:

1. Seznamte se s problematikou překladačů, konrétně přečtením odborné literatury a prostudováním existujících Open Source překladačů.
2. Seznamte se s knihovnou Eigen pro C++, a předveďte v ní základní funkce jazyka Octave.
3. S použitím platformy github a Travis CI vytvořte prostředí, ve kterém budete překladač implementovat a testovat
4. Implementujte překladač, který zvládá všechny základní funkce jazyka Octave přeložit do zdrojového kódu C++ s Eigen
5. Diskutujte dosažené výsledky a možnosti pokračování práce.

Literatura:

- https://www.gnu.org/software/octave/doc/interpreter/Simple-Examples.html
- http://eigen.tuxfamily.org/dox/
- https://docs.hhvm.com/hhvm/
- http://www.abclinuxu.cz/clanky/programovani/jazyky-a-prekladace-1-uvod

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3 zadání

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kolář Martin, M.Sc.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Programy vyvinuté v interaktivním programovém prostředí Matlab je náročné využít na zařízeních s malým množstvím paměti a v integraci do projektů bez podpory tohoto jazyka. Proto jsou programy převáděny do jazyka C++. V praxi se používá manuálního převodu, který výrazně prodlužuje dobu nasazení. Tato práce se zaměřuje na automatizování překladu z jazyka Octave/Matlab do C++ s použitím knihovny Eigen umožňující využití maticových a vektorových operací. Překladač umožňuje překlad základních 39 operací a 13 funkcí jazyka Octave. Experimenty ukazují, že se tímto překladem dosáhne snížení požadavku paměti až o 99%.

Abstract

It is difficult to use programs developed in the interactive programming environment Matlab for low-memory devices and for integration into projects without the language support. Therefore, the programs are converted into C++. In practice, manual transfer is used which significantly prolongs the time of the developing program. The work focuses on the automation of translation from Octave/Matlab to C++ using the Eigen library to enable matrix and vector operations. The translator allows 39 basic operations and 13 functions of the Octave language. Experiments show that this translation will reduce memory requirements of up to 99%.

Klíčová slova

Překladač, transpiler, kompilátor, Eigen, Octave, C++, generování kódu, paměťová náročnost, matice

Keywords

Compiler, transpiler, transcompiler, source-to-source compiler, Eigen, Octave, C++, code generation, memory requirement, matrices

Citace

ŠEVČÍK, Václav. *Překladač z Octave do C++*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Kolář Martin.

Překladač z Octave do C++

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Martina Koláře M.Sc. Další informace mi poskytl Ing. Zbyněk Křivka, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Václav Ševčík
13. května 2017

Poděkování

Děkuji vedoucímu práce Martinu Kolářovi M.Sc. a Ing. Zbyňku Křivkovi, Ph.D., kteří mi poskytli odbornou pomoc při tvorbě překladače a psaní bakalářské práce. Děkuji též lidem, kteří vytvořili LaTeXovou šablonu, jež mi ulehčila psaní bakalářské práce. Hlavní dík patří Bohu, který mě provází po celou dobu studia. Poděkování náleží také mé rodině, která mi poskytovala psychickou a materiální podporu.

Obsah

Kapitola 1

Úvod

Množství jazyků, zaměřených na matematické výpočty, využívá k vyhodnocení interpret[1], což umožňuje zvýšit rychlost práce díky poskytnutým vestavěným funkcím a vyšší abstrakci (netypovost, asociativní pole, atd.) těchto jazyků. Vývoj zmíněným způsobem je výhodný zejména z hlediska času a práce programátora. Pro platformy s malou kapacitou paměti je však toto řešení příliš paměťově náročné (např. vestavěné systémy). Jedním z možných řešení je převést program do jiného jazyka, který nevyžaduje paměť na uložení objemného interpretu. Překladem jsou ponechány výhody při vývoji ve výše zmíněných jazycích.

Pomocí překladače (transpileru) můžeme převést zdrojový kód interpretovaného jazyka (Octave/Matlab) do zdrojového kódu kompilovaného jazyka (C++). Tento zdrojový kód se přeloží do podoby binárního kódu cílové platformy, na kterou může být poté nahrán. Překlad může být proveden na jiné než cílové platformě s dostatkem paměti a možností překladu kompilovaného jazyka (C++). Tímto postupem můžeme docílit toho, že výpočty mohou běžet na platformách, kde by nemohl být uložen a spuštěn interpret (Octave/Matlab).

Cílem této práce je vytvořit transpiler, který umožní převod podmnožiny jazyka Octave do jazyka C++, konkrétně s využitím knihovny Eigen, za účelem úspory paměti (absence interpretu) a zvýšení přenositelnosti programu. Přenositelností mezi jazyky je v tomto kontextu míněna například možnost integrace programů z Octave do frameworku OpenCV. Překladač je psán objektově orientovaně v jazyce C++ s možností snadné rozšiřitelnosti o další funkcionalitu jazyka Octave.

Transpilery se často využívají v praxi k překladu z různých jazyků (PHP, C++, Pascal) do jazyka C kvůli jeho podpoře na většině platforem a též jeho rychlosti. Taktéž se využívají k překladu skriptovacích jazyků jako jsou CoffeeScript, TypeScript do JavaScriptu (viz transpiler [14]), který je v této oblasti často využíván a je též hojně podporován ze strany programů.

V 2. kapitole práce se vysvětluje teorie překladačů, význam a jejich obvyklá struktura. Je zde popsáno od základních principů překladačů přes jednotlivé překladové fáze až po nejčastější metody tvorby gramatiky překladačů. V poslední podkapitole jsou představeny jednotlivé nástroje pro tvorbu překladačů.

3. kapitola přibližuje technologie, které jsou pro výstavbu řešeného překladače nezbytné. Představuje jazyk Octave a knihovnu Eigen. Ukazuje, v čem jsou si tyto jazyky podobné, ale též v čem jsou jejich rozdíly, s nimiž je nutné při implementaci počítat.

Po seznámení se s problematikou překladačů a použitých technologii následuje návrh vytvořeného překladače v kapitole 4. Kapitola začíná odůvodněním zvoleného postupu im-

[1] Program pro vyhodnocování zdrojového kódu v reálném čase.

plementace překladače. V druhé podkapitole ukazuje strukturu překladače pomocí UML diagramu. V poslední podkapitole jsou dekomponovány jednotlivé fáze překladu do jednotlivých tříd, které budou zodpovídat za danou funkcionalitu.

5. kapitola vysvětluje vlastní implementaci překladače. V jednotlivých podkapitolách je zde vysvětleno, jak výsledný překladač je implementován a jak se postupovalo v řešení jednotlivých problémů.

Poslední 6. kapitola obsahuje návod k obsluze pro budoucí uživatele. Zpočátku se zabývá způsobem instalace překladače, následuje příklad způsobu užití, nutné podmínky pro použití překladače a automatické testování funkčnosti překladače pomocí předem vytvořené testovací sady. V poslední podkapitole je poukázáno na hlavní výhody vytvářeného překladače a popsány důvody k jeho použití.

Hlavní úspěch této práce spočívá ve vytvoření překladače, který dokáže ulehčit portaci vědeckého software na platformu pro aplikované využití. Tento překladač pomůže k rychlejšímu nasazení software v praxi.

Kapitola 2

Překladač

Překladač je program, který čte program napsaný ve zdrojovém kódu a překládá ho do ekvivalentního programu v cílovém kódu. Typicky překládá jazyk na vyšší úrovni abstrakce do jazyka symbolických adres, které jsou specifické pro danou architekturu procesoru. Z jazyka symbolických adres si už procesor ve své režii vygeneruje strojový kód.

Tato kapitola vysvětluje výstavbu překladačů. Začíná od jejich definic v podobě základních pojmů formálních jazyků a pokračuje až po jednotlivé algoritmy, které jsou využívány při implementaci překladačů.

První podkapitola 2.1 pojednává o základních teoretických principech a základních pojmech formálních jazyků, o kterých bychom při vytváření překladače měli vědět. Dále se zabývá podrobněji bezkontextovými jazyky, které bude překladač zpracovávat. Poslední podkapitola poukazuje na rozdíl mezi překladačem a transpilerem. Podkapitola čerpá z 1. a 5. kapitoly knihy [17] a z přednášek předmětu Formální jazyky a překladače vyučovaném na FIT VUT v Brně [20].

Druhá podkapitola 2.2 se zaměřuje na vysvětlení, jak funguje překladač, pomocí rozdělení překladu na jednotlivé fáze. Překlad začíná vždy lexikální analýzou, pokračuje hlavní fází překladače syntaktickou analýzou, která spolupracuje se sémantickou analýzou. Po fázích analýzy se může generovat vnitřní kód. Následuje volitelná fáze optimalizace a jako poslední generování cílového kódu. Tato podkapitola vybírá informace z knihy [18] z kapitol 2, 3, 6 a 7 a taktéž z knihy [1], konkrétně 3.-8. kapitoly.

Třetí podkapitola 2.3 ukazuje možné způsoby implementace překladačů pomocí různých algoritmů. V první sekci jsou ukázány analýzy, které vytvářejí syntaktický strom shora dolů. V druhé se vysvětluje opačný postup, když se vyhodnocuje zdola nahoru. Informace k této podkapitole pochází z knih: [3] 16.-18. kapitoly, [18] 4. a 5. kapitola, z [19] 6. a 7. kapitoly, a též ze skript [28].

Poslední podkapitola 2.4 ukazuje různé prostředky pro vývoj překladače. Jsou zde popsány nástroje, které mohou generovat jednotlivé fáze pomocí zápisu kódu ve své syntaxi, většinou formou pravidel, či regulárních výrazů. Poznatky o jednotlivých nástrojích jsou získány ze stránek vývojářů těchto nástrojů, konkrétně ze stránek [16], [12], [2] a [22]. V sekci se představují též nástroje pro tvorbu transpilerů, o kterých jsou informace čerpány ze stránek [23] a [13].

2.1 Úvod do formálních jazyků

Výstavba samotného překladače vychází z teorie formálních jazyků. Dříve, než pokročíme k jednotlivým fázím překladu, je vhodné vysvětlit základní pojmy, které jsou pak oporou při výstavě překladače.

První část vysvětluje základní pojmy formálních jazyků 2.1.1 formou definic. Dále je podrobněji zaměřeno na bezkontextové gramatiky 2.1.2 a jejich vlastnosti. Definice z těchto částí jsou převzaty z knihy [20]. V poslední části je představena speciální varianta překladače – transpiler a její zasazení do celkového kontextu rodiny překladačů 2.1.3. Hlavním zdrojem porovnání jazyků je článek [15].

2.1.1 Základní pojmy formálních jazyků

Před vysvětlením, co je to jazyk 2.1.1.3 v kontextu formálních jazyků, je nutné vysvětlit pojmy, ze kterých vychází, a to abecedu 2.1.1.1 a řetězec 2.1.1.2. Pro zjednodušení terminologie jsou zde vynechány operace a funkce nad jazyky a řetězci.

Definice 2.1.1.1. Abeceda je konečná, neprázdná množina elementů, které nazýváme symboly.

Definice 2.1.1.2. Nechť Σ je abeceda 2.1.1.1.
1) ε je řetězec nad abecedou Σ, kde ε = prázdný řetězec
2) pokud x je řetězec nad Σ a $a \in \Sigma$, potom xa je řetězec nad abecedou Σ

Definice 2.1.1.3. Nechť Σ^* značí množinu všech řetězců 2.1.1.2 nad Σ. Každá podmnožina $L \subseteq \Sigma^*$ je jazyk nad Σ.

Každý jazyk 2.1.1.3 v Chomského hierarchii jazyků[1] lze popsat gramatikou 2.1.1.4, která je založena na konečné množině gramatických pravidel, které generují řetězce výsledného jazyka. Po sestavení gramatiky 2.3 ji lze použít pro překlad celého vstupního řetězce na celý výstupní řetězec.

Gramatika dokáže popsat, jak konečné, tak i nekonečné jazyky. Sjednocením abeced terminálů a neterminálů dostáváme slovník gramatiky. Neterminály z gramatiky se postupně pomocí derivací rozkládají na řetězce terminálů a neterminálů.

Definice 2.1.1.4. Gramatika je čtveřice $G = (N, T, P, S)$, kde

- N je abeceda neterminálů
- T je abeceda terminálů
- P je konečná množina pravidel pro přepis
- $S \in N$ je počáteční neterminál

V kontextu této kapitoly překladač definujeme jako program, který překládá zdrojový kód v jednom jazyku 2.1.1.3 do cílového kódu druhého jazyka. Vstupní a výstupní kód je sémanticky ekvivalentní. Když v programu je nalezena chyba, analýza to oznámí uživateli. Poté se může pokusit zotavit z chyby, anebo neúspěšně skončí překlad.

[1] hierarchie tříd jazyků generujících formální jazyky

2.1.2 Bezkontextové jazyky

Definice 2.1.2.1. Nechť L je jazyk 2.1.1.3. L je bezkontextový jazyk (BKJ), pokud existuje bezkontextová gramatika, která generuje tento jazyk L.

Bezkontextová gramatika vychází z definice gramatiky 2.1.1.4, ale přidává si upřesňující podmínky pro dvě komponenty ze čtveřice.

- T je abeceda terminálů, přičemž $N \cap T = \emptyset$
- P je konečná množina pravidel tvaru $A \to x$, kde $A \in N$, $x \in (N \cup T)^*$

Což znamená, že oproti regulární gramatice může derivací[2] generovat posloupnost, kde terminály a, c jsou na obou stranách vzniklého neterminálu: $A \to aBc$. Tímto způsobem dokáže přijímat i jazyky s závorkovými strukturami (např. ve výrazech) a zkontrolovat jejich správné párování.

Pro bezkontextové jazyky jsou dva ekvivalentní fundamentální modely:

- bezkontextové gramatiky – lze zapsat různými formami: BNF[3], EBNF[4], syntaktické grafy
- zásobníkové automaty – konečný automat rozšířený o zásobník

Při zpracování bezkontextových jazyků v překladačích se nejčastěji využívají zásobníkové automaty 2.1.2.2.

Definice 2.1.2.2. Zásobníkový automat (ZA) je sedmice $M = (Q, \Sigma, \Gamma, R, s, S, F)$, kde

- Q je konečná množina stavů
- Σ je vstupní abeceda
- Γ je zásobníková abeceda
- R je konečná množina pravidel tvaru $Apa \to wq$, kde $A \in \Gamma$, $p, q \in Q$, a $\in \Sigma \cup \{\varepsilon\}$, $w \in \Gamma^*$
- $s \in Q$ je počáteční stav
- $S \in \Gamma$ je počáteční symbol na zásobníku
- $F \subseteq Q$ je množina koncových stavů

Inicializovaný (na zásobníku pouze počáteční symbol) zásobníkový automat pomocí symbolů na vrcholu zásobníku a na vstupu provede přechod. Pokud je na vrcholu zásobníku terminál, zkontroluje, jestli odpovídá vstupu. Při shodě terminálů se odstraní terminál z vrcholu zásobníku. Pokud je na vrcholu neterminál, odstraní se vrchní symbol a nahradí se symboly určenými pravidly přechodu. Pokud nenastala chyba při přechodu (špatné symboly na jednom ze vstupů), pokračuje přečtením dalšího symbolu ze vstupu a celý proces se opakuje. Úspěšné přijetí celého řetězce v zásobníkovém automatu může nastat následujícími třemi způsoby: při vyčerpání zásobníku, přechodem do koncového stavu, nebo když nastanou obě události.

[2] Přepis neterminálu na řetězec
[3] Backusova-Naurova forma = metasyntaxe používaná k vyjádření bezkontextové gramatiky
[4] Rozvinutá Backusova–Naurova forma – rozšiřuje BNF o možnost opakování a volitelnosti výrazů

2.1.3 Vazba mezi překladačem a transpilerem

Transpiler je speciální typ překladače, který je specifický hlavně vysokou mírou abstrakce výstupního jazyka. Ale podstata překladače nemizí. Překládá zdrojový kód jednoho jazyka do zdrojového kódu jiného jazyka. Strukturou jednotlivých fází překladu se transpiler neliší od jiných překladačů.

Mezi hlavní výhody transpileru patří:

- Může přeložit zdrojový kód do jazyka, který pro překlad může využít výhody cílového překladače
- Generuje strukturovaný, čitelný vnitřní kód. Je proto snadné převést ho do jiného jazyku pomocí jednoduchého překladače
- Ušetří čas strávený manuálním překladem z jednoho jazyka do druhého
- Větší konstrukce mohou být nahrazeny kratšími díky převodu do jazyka s větší abstrakcí [15]

Tento typ překladače se často využívá na překlad do JavaScriptu z různých skriptovacích jazyků nebo též na překlad z již moc nepoužívaných jazyků jako Fortran či Pascal do jazyka C.

2.2 Fáze překladu

Překladač dělíme na dva fundamentální celky, a to na analýzu kódu programu a na syntézu cílového programu. Každá z těchto komponent se skládá z dílčích částí, které zodpovídají za jednotlivé úkoly v běhu překladače. O každé z těchto částí je pojednáno v této podkapitole. Obrázky v této podkapitole jsou převzaty a upraveny z obrázku v [1] ze strany 7.

Analýza kódu programu se skládá z lexikální analýzy 2.2.1, která předpřipraví zdrojový kód do logických částí – tokeny. Dále syntaktická analýza 2.2.2 prověří správnou syntaxi programu. Spolupracuje přitom se sémantickou analýzou 2.2.3, která kontroluje správnou sémantiku, například typovou správnost jednotlivých výrazů.

Po zkontrolování správnosti programu a jeho dekompozici přichází na řadu syntéza cílového programu. V prvním kroku si většinou vyrobí svůj vnitřní kód 2.2.4, který posléze slouží pro lepší možnost optimalizace 2.2.5. V poslední fázi je optimalizovaný vnitřní kód převeden na cílový výstupní kód 2.2.6.

2.2.1 Lexikální analýza

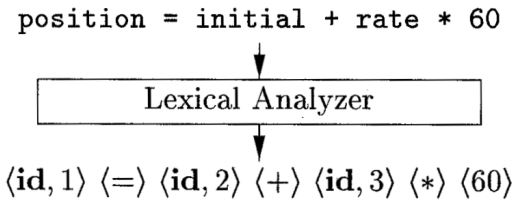
Lexikální analýza čte po jednotlivých znacích zdrojový kód a vytváří z nich smysluplné konstrukce nazývané lexémy. Tímto kontroluje, jestli přicházejí syntakticky správné elementy vstupního jazyka. Pokud nedokáže rozpoznat vstupní posloupnost znaků, zahlásí chybu lexikálního analyzátoru. Tyto lexémy pak předává dál zaobaleny v tokenech, které mají následující formu:

$$< nazev\_tokenu, hodnota\_atributu >$$

Kde název_tokenu je abstraktní symbol, který je později použit v syntaktické analýze a nese informaci o typu tokenu. Hodnota_atributu ukazuje na místo v tabulce symbolů 2.2, kde je umístěna hodnota lexému.

Tokeny se posílají na požádání syntaktické analýze. Lexikální analyzátor mimo jiné může zjednodušovat zpracování vstupního kódu nahrazováním, či odstraňováním nadbytečných bílých znaků nebo komentářů.

Příklad můžeme vidět na obrázku 2.1.

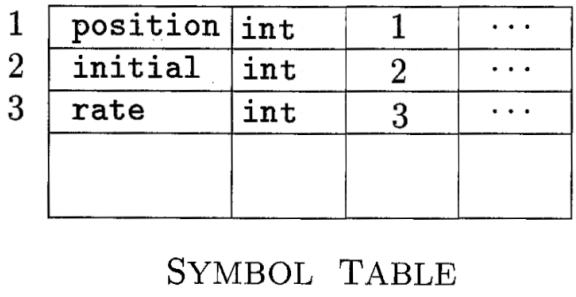


Obrázek 2.1: Rozčlenění vstupních znaků do tokenů.

Existují dva modely k řešení lexikální analýzy:

- Regulární výrazy – reprezentují jednoduchý způsob, jak zapsat jazyk do formulí. Regulární výrazy jsou založené na používání operací konkatenace, spojení a iterace, které umožňují obsáhnout celou třídu regulárních jazyků.
- Konečné automaty – způsob zpracování příchozích znaků jazyka do lexémů. Přijme znak, provede přechod do dalšího stavu. Pokud již není přechod ze stavu možný, podívá se, zda je v koncovém stavu. Pokud ano, příjme lexém, jinak nastává chyba. Poté načítá další znak a postup se opakuje z počátečního stavu.

Tabulka symbolů nás bude provázet přes celý překlad. Uchovávají se v ní názvy proměnných a též názvy funkcí. Tabulka poskytuje informace o jednotlivých tokenech pro sémantickou analýzu a generování kódu. Způsob organizace záznamů v tabulce zásadně ovlivňuje rychlost přístupu ke čtení a zápisu. Pro menší rozsahy počtu záznamů si můžeme vystačit s polem či jednosměrně vázaným seznamem, ale pro větší počty záznamů se již vyplatí použít binární vyhledávací stromy nebo hašovací tabulky. Jednotlivý záznam se skládá ze jména tokenu a vlastností. Příklad takové tabulky můžeme vidět na obrázku 2.2.



Obrázek 2.2: Umístění informací tokenů do tabulky symbolů.

2.2.2 Syntaktická analýza

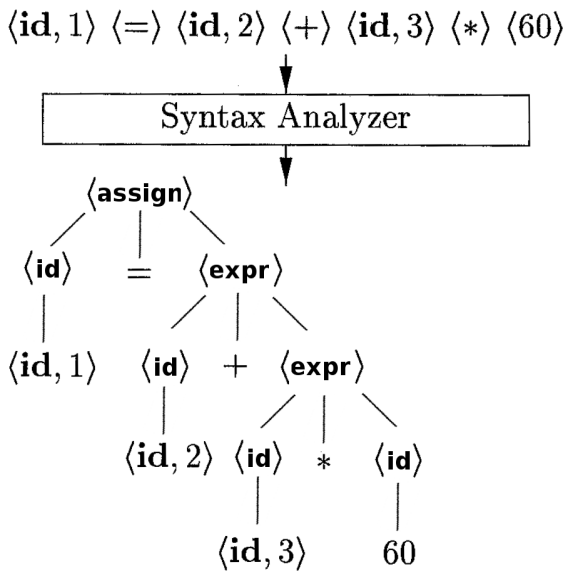
Syntaktická analýza je postavená na gramatice (resp. zásobníkovém automatu) 2.1.2, která je tvořena konečným počtem pravidel. Pomocí pravidel kontroluje správnost syntaxe —

posloupnosti přijímaných tokenů z lexikální analýzy. Tato kontrola probíhá způsobem procházení vstupního řetězce zleva doprava, kde se kontroluje shodnost přijímaných terminálů s očekávanými terminály. Pokud je v pravidlu neterminál derivuje se podle vhodného gramatického pravidla na posloupnost terminálů a neterminálů. Pokud toto pravidlo nenajde nastane chyba překladu, kvůli nečekané syntaktické konstrukci.

Derivace se dá graficky vyjádřit pomocí derivačního stromu 2.3. Kde vztahy mezi rodičem a dětmi reprezentují gramatická pravidla. Zpracování jedné struktury můžeme vyjádřit doplněním tokenů do jednotlivých uzlů syntaktického stromu.

Syntaktická analýza je úzce spjata s analýzou sémantickou, s níž často bývá považována za hlavní centrum řízení překladače. Tento způsob překladu se nazývá syntaxí řízený překlad. Z této části se volají další analýzy a jednotlivé další postupy v překladu.

Pro konstrukci syntaktické analýzy existují dva druhy metod, a to metoda shora dolů a zdola nahoru. Jak název napovídá zpracování probíhá abstrakcí zdola nahoru, nebo konkretizací shora dolů. Podrobněji se těmito metodami zaobírá podkapitola 2.3.



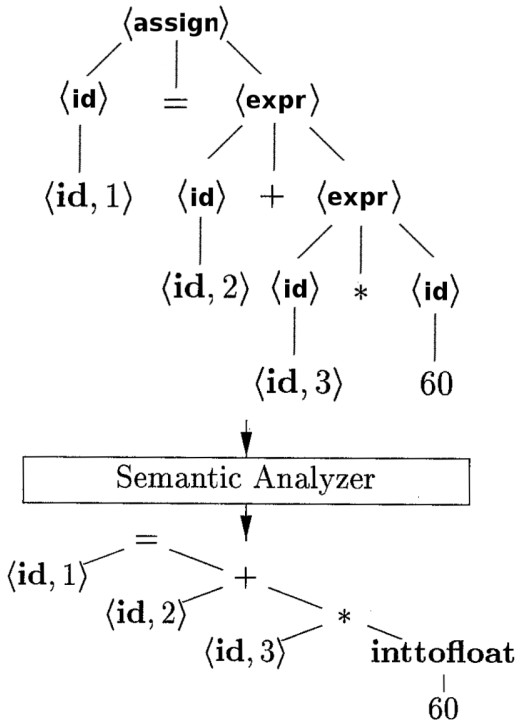
Obrázek 2.3: Převod do derivačního stromu.

2.2.3 Sémantická analýza

Sémantická analýza je volána funkcemi ze syntaktické analýzy, z níž dostává derivační strom. Nad tímto stromem provádí kontroly správnosti datových typů a případné implicitní typové konverze. Informace o jednotlivých proměnných a funkcích získává z tabulky symbolů (viz obrázek 2.2). Příklad implicitního přetypování můžete vidět na obrázku 2.4.

Též kontroluje skokové instrukce měnící běh programu a zda v programu existují korespondující návěští. Dále hlídá, aby názvy proměnných a funkcí byly unikátní. V neposlední řadě zajišťuje kontrolu, jestli je daná proměnná definovaná, nebo zda již není po své době životnosti.

Z této fáze překladu může vycházet buďto abstraktní syntaktický strom, ve kterém jsou zaznamenány jednotlivá pravidla, nebo již vytváří společně s syntaktickou analýzou vnitřní kód.



Obrázek 2.4: Implicitní přetypování u sémantické analýzy.

2.2.4 Generování vnitřního kódu

Během procesu překladu mezi zdrojovým a cílovým kódem si může překladač vytvořit vlastní kód pro lepší práci s kódem programu. Děje se tak hlavně po syntaktické a sémantické analýze, kdy si většinou překladač vytvoří nízko-úrovňový vnitřní kód podobný kódu symbolických instrukcí. Kód musí splňovat dvě podmínky: musí být snadno sestavitelný z předchozího kódu a musí být lehce přeložitelný do cílového kódu. Pro tyto účely se často využívá tříadresný kód, který vyhovuje těmto kritériím.

Tříadresný kód reprezentuje jednoduché instrukce, a proto se snadno překládá do kódu symbolických instrukcí. Při jeho vytváření mají jednotlivé pozice čtveřice doporučený význam.

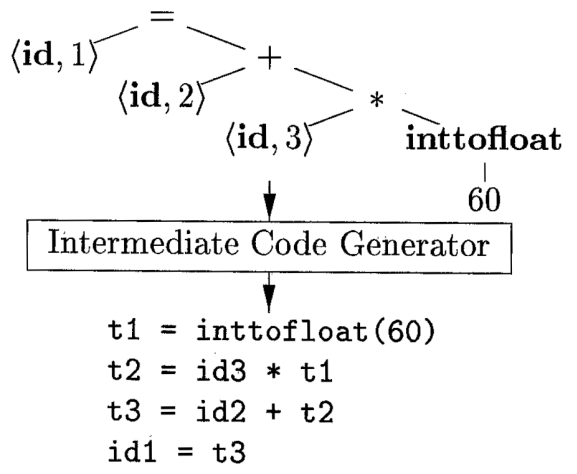
$$[O, X, Y, Z]$$

Na první pozici je operátor O, následují dvě pozice pro operandy X *a* Y a nakonec cíl Z, do kterého se ukládá výsledek. Pro vnitřní reprezentaci si tříadresný kód generuje vlastní proměnné, které poté používá pro provádění výpočtů. Generování dočasných proměnných je vyžadováno kvůli převodu do nižšího stupně abstrakce. Tím pádem v kódu musí být použito více kroků k výpočtu komplexnějších příkazů.

Vygenerováním vnitřního kódu získáváme tyto výhody:

- Vnitřní kód usnadňuje optimalizaci.
- Zjednodušuje množství různých konstrukcí pouze na omezené množství elementárních instrukcí.
- Usnadňuje generování cílového kódu.

Převod abstraktního syntaktického stromu do vnitřního kódu můžeme vidět na obrázku 2.5.



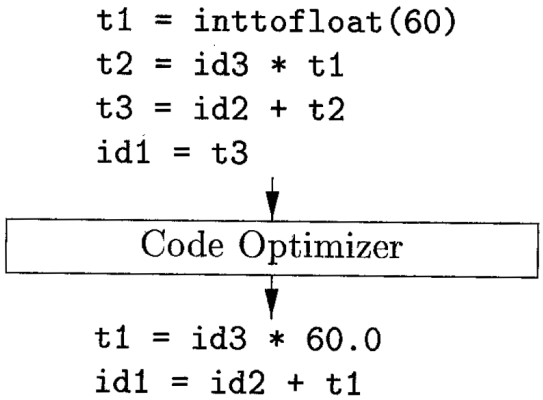
Obrázek 2.5: Generování vnitřního kódu z abstraktního syntaktického stromu.

2.2.5 Optimalizace

Optimalizace je volitelná fáze, která umožňuje urychlení běhu vytvářeného programu. Stávající kód můžeme optimalizovat vynecháním částí kódu, který neovlivňuje zbytek programu, nebo nahrazení složitějších programových konstrukcí jednoduššími. Výsledkem této fáze není vyrobit optimální kód, ale vylepšit stávající. Při překladu si lze vybrat, zda chceme optimalizaci použít nebo ne. Vypnutí optimalizací totiž výrazně zkrátí dobu překladu.

Optimalizace probíhá po blocích kódu, ve kterých není žádné návěští ani skok – nenarušuje tok programu. Tím pádem operace nad určitým blokem kódu neovlivní zbytek programu. Optimalizovat lze jak základní věci jako je šíření konstanty programem, šíření kopírováním (zajišťuje nejmenší počet kopírovaní hodnot při přenášení do cílové proměnné) a eliminace mrtvého kódu (část programu, ke které se nemůže žádným způsobem dostat tok programu), tak i komplexnější věci, které kvůli své složitosti vyžadují více času.

Optimalizace vnitřního kódu můžeme vidět na 2.6.

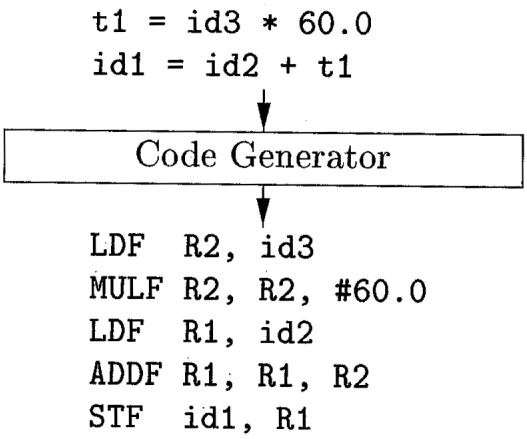


Obrázek 2.6: Optimalizace vnitřního kódu.

2.2.6 Generování cílového kódu

Generování cílového kódu stojí na konci posloupnosti překladových fází překladače. Hlavním kritériem této fáze je správnost výsledného kódu a také splnění syntaxe cílového jazyka. Program generovaný překladačem musí být funkčně ekvivalentní vůči vstupnímu programu do překladače. Nejčastěji překládá do jazyka nízké abstrakce — kódu symbolických instrukcí, ale může generovat i kód na stejné míře abstrakce 2.1.3.

Vstupem může být vnitřní kód nebo syntaktický abstraktní strom (např. výrazy ve formě polské notace), ze kterého generuje kód v cílovém jazyce. Generování cílového kódu můžeme vidět na obrázku 2.7.



Obrázek 2.7: Generování cílového kódu v assembleru.

2.3 Používané přístupy pro tvorbu derivačních stromů

Podkapitola pojednává o používaných přístupech pro tvorbu derivačních stromů pro syntaktickou analýzu 2.2.2. Přístup umožňuje využít gramatiku pro kontrolu správnosti syntaxe. Vytváření gramatik se dělí na dva typy podle způsobu vytváření derivačního stromu. Buďto se začíná od jednotlivých tokenů a sestavuje se derivační strom redukcemi směrem nahoru ke kořenu 2.3.2, nebo postupnými derivacemi rozvineme derivační strom od kořene až na jednotlivé listy 2.3.1, kterými jsou tokeny. Pro oba způsoby je společné, že prochází a zpracovávají vstupní řetězec tokenů zleva doprava.

2.3.1 Analýza shora dolů

Metoda zpracovává řetězec od kořene až po listy. Začíná derivací kořenu a pokračuje s nejlevější derivací až do doby, než je na levé straně terminál, který se porovná se vstupním tokenem. Poté pokračuje na další uzel v derivačním stromu. Pro tento přístup je charakteristické, že vybírá vždy nejlevější derivace daného stromu.

Nejčastěji se tato metoda používá nad LL gramatikami. Typicky se zapisuje do LL tabulky, která má na řádcích neterminály a ve sloupcích terminály, které mohou být derivovány. Pravidla jsou zapsána do buněk, které určují, jestli derivací neterminálu může vzniknout na prvním místě v řetězci daný terminál. O implementaci LL tabulky viz [27]. Gramatiku lze považovat za LL gramatiku, pokud existuje pouze jedno pravidlo pro derivaci z daného neterminálu pomocí terminálu na vstupu. Množina LL gramatik je podmnožinou

bezkontextových gramatik. Tento přístup se používá, když se překladač implementuje bez pomocí nástrojů.

Příklady implementací prediktivní analýzy

- S využitím zásobníku – Tato implementace čerpá informace pro vyhodnocení z LL tabulky, ve které jsou zaznamenána jednotlivá pravidla derivace. Vzniklé symboly z jednotlivých derivací se nahrávají na zásobník. Hlavní nevýhodou je nutná implementace zásobníku. Tento druh implementace ale poskytuje lehce modifikovatelnou implementaci LL gramatiky, kde není problém pozměnit, doplnit či odebrat jednotlivá pravidla.
- Rekurzivní sestup – Implementace staví na rekurzivním volání v sobě zanořených jednotlivých funkcí. Děje se tak podle pravidel derivací. Hlavní výhoda této implementace je, že není zapotřebí implementovat zásobník. Protože změna požadavků může způsobit nutnost přepsání více funkcí, používá se tato implementace přednostně tam, kde jsou pevně daná syntaktická pravidla.

2.3.2 Analýza zdola nahoru

V přístupu se začíná od jednotlivý listů a pomocí redukcí a přesunů (angl. shift) se snaží zredukovat listy na kořen. Přitom vytváří nejpravější derivační strom. Tento způsob sestavování derivačního stromu má větší sílu — zahrnuje větší množinu než LL gramatiky. Proto se hojně používá na vytváření překladačů. Nejčastěji pomocí pomocných nástrojů na tvorbu překladačů.

Symboly se nahrávají na zásobník, kde se na ně aplikují překladová pravidla pro redukci. Pokud není možná redukce, pokračuje se s načítáním dalších tokenů. Vytváří se tak nejpravější derivační strom. Tento způsob syntaktické analýzy využívají: precedenční analýza operátorů a LR gramatika.

Precedenční analýza operátorů — využívá se při vyhodnocování správnosti priority operátorů. Nahrává si jednotlivé výrazy na zásobník, kde je podle pravidel, které jsou umístěné ve vlastní tabulce priority operátorů, redukuje. Pokud dokáže zredukovat celý výraz na počáteční neterminál, zajistí správnou interpretaci všech výrazů. Řeší též asociativitu operátorů, která určuje, z jakého směru je operátor vyhodnocován. Tato metoda nemá příliš vysokou vyjadřovací sílu, kvůli problémem s operátory s aritou jinou než binární, ale využívá se kvůli snadné implementaci. Využívá se proto právě na vyhodnocení výrazů tam, kde není tolik potřebná vyjadřovací síla.

LR syntaktická analýza — využívá se na výstavbu syntaktických analyzátorů kvůli několika nesporným výhodám oproti LL analýze. Její rozsah přijímaných jazyků (síla) je takový, že přijímá všechny deterministické bezkontextové jazyky, což je více než u jiných analýz. Nezmenšuje rychlost překladu a dokonce v některých případech šetří výpočetní výkon tím, že některá pravidla vyhodnotí dříve, než se dostanou na zásobník. Výhodou je též elegantní vyhledání syntaktické chyby v programu. Jedinou její nevýhodou je její složitější tvorba. Proto se k ulehčení tvorby analyzátorů využívají nástroje.

2.4 Prostředky pro tvorbu překladačů

Překladač nemusí být vybudováván vždy celý od začátku, programátor si může pomoci nástroji, které mu pomohou při sestavení jednotlivých fází překladače. Následující odstavce představí několik nástrojů, které ulehčují práci programátorům snadnějším a přehlednějším zápisem jednotlivých fází.

Lex [16] — je nástroj pro tvorbu lexikální analýzy. Předpisy pro jednotlivé lexémy lexikální analýzy (viz podkapitola 2.2.1) se tvoří pomocí regulárních výrazů ve stejnojmenném jazyku Lex. Zdrojový kód se překládá pomocí překladače Lexu do jazyka C. Tento vzniklý modul lze použít jako lexikální analyzátor.

Yacc [12] — slouží pro výstavbu syntaktického analyzátoru, který hlídá správnou syntaxi. Používá metodu LALR, která sice dokáže postihnout menší podmnožinu bezkontextových jazyků než LR, ale je upřednostňovaná kvůli úspoře paměti [11].

PLY [2] — kombinuje nástroje Lex a Yacc. Poskytuje funkce těchto nástrojů pro jazyk Python. Velkou výhodou je stálý vývoj tohoto nástroje. Očekávaná funkcionalita je tedy sjednocením dvou výše uvedených nástrojů.

ANTLR [22] — je nástroj pro čtení, zpracování, provádění a přeložení strukturovaného textu nebo binárního souboru. Nejčastěji se používá pro překlad jazyků. Z gramatiky v EBNF[5] dokáže ANTLR vytvořit všechny druhy (lexikální, syntaktický a predikátní sémantický) analyzátorů.

2.4.1 Prostředky pro tvorbu transpileru

Existují i úzce specializované nástroje, které se zaměřují konkrétně na jeden typ překladačů a to na transpilery. Tyto nástroje se vyznačují širokou funkcionalitou a poskytují hodně možností pro využití.

ROSE [23] — je svobodně šiřitelný transpiler (viz podkapitola 2.1.3), který poskytuje transformaci a analýzu širokého spektra jazyků. Konkrétně C (C89 a C98), C++ (C++98 a C++11), UPC, Fortran (77/95/2003), OpenMP, Java, Python a PHP.

Pro tvorbu v nástroji ROSE by měli mít uživatelé zkušenosti s tvorbou překladačů (vědci zabývající se překladači nebo vývojáři knihoven a nástrojů s zkušenostmi s překladači). ROSE je vhodný pro statickou analýzu, optimalizaci programu, libovolnou transformaci programu, doménově specifické optimalizace, analýzu výkonnosti a optimalizaci cyklů.

Reprezentace vnitřního kódu v nástroji ROSE má vysokou míru abstrakce charakteristickou pro transpilery, ze které se následně sestavuje abstraktní syntaktický strom. Nástroj navzdory převodu do vnitřního kódu s vysokou mírou abstrakce neztrácí žádné informace o struktuře původního zdrojového kódu.

V tomto nástroji byla již vyvinuta řada překladačů. Vytvářené nástroje využívají aplikační rozhraní transpileru, které poskytuje základní funkce pro práci s ním. Nástroj dovoluje zahrnout velké množství různých analýz: analýza grafu volání, analýza toku řízení, analýza

[5] Extended Backus–Naur Form — Rozvinutá Backusova-Naurova forma

toku dat (život proměnných, dvojice def-use, dosažitelnost definice, atd.), analýza hierarchie tříd, analýza závislostí dat a systému a analýza vzorů komunikace.

Nástroj ROSE též umožňuje velké množství druhů optimalizací: *partial redundancy elimination, constant folding, inlining, outlining, OpenMP directive lowering,* automatická paralelizace a transformace cyklů (optimalizace cyklů podporuje agresivní optimalizace jako jsou slučování, záměna, invariant a rozbalení cyklů).

ROSE je vydána pod BSD licencí s podporou operačních systémů Linux a Mac OS X. Používá se pro rozličné druhy výzkumu, vývoje a též k studijnímu využití.

Meta-enviroment [13] — je framework[6] pro vývoj transpilerů. Poskytuje nástroje pro syntaktickou analýzu, sémantickou analýzu a generování výstupního kódu s interaktivním vývojovým prostředím. Nástroj je šířen jako svobodný software. Snadno se modifikuje a je rozšiřitelný moduly (angl. plugin) od třetích stran.

V současné době se používá nejčastěji na:

- Syntakticky rozbor programovacích jazyků pro pozdější zpracování v syntaktických stromech
- Analýzu zdrojového kódu (typová analýza a generování dokumentace)
- Transformace, přeuspořádání a generování kódu

Meta-enviroment se vyznačuje možností modulárně zadat definici gramatiky. Poskytuje deklarativní filtr pro vyřešení většiny nejednoznačností v programovacích jazycích. Nabízí navíc vestavěné vývojové prostředí, které je interaktivní a pomáhá v práci s nástrojem.

Tento nástroj se úspěšně používá na analýzu SQL dotazů a databázových schémat, přeuspořádání jazyka Cobol, PL/I syntaktickou analýzu, restrukturalizaci C++, hledání mrtvého kódu a hledání nezvyklých konstrukcí (*smell detection*) v jazyce Java. [5]

K sestavení překladače potřebuje nástroj dvě formální definice, a to definici syntaxe a definici algebraických operací. Nástroj nevyužívá lexikální analýzu oddělenou od syntaktické, ale syntaktická analýza integruje lexikální analýzu, což způsobuje zrychlení analýzy. Nástroj používá pro LR analýzu nástroje Yacc a Bison. [4]

[6] Softwarové prostředí, které poskytuje programátorovi výhody při vytváření jiných programů (např: knihovny, podpůrné podprogramy, návrhové vzory, atd.)[26]

Kapitola 3

Technologie

Tato kapitola pojednává o technologiích, které byly použity v práci na transpileru. Prvně je zde popsán jazyk Octave 3.1, který slouží jako vstupní jazyk do transpileru. Druhá podkapitola se zabývá C++ knihovnou Eigen 3.2, která se používá pro generování výstupního kódu z transpileru. Pro psaní překladače je nezbytné zjistit rozdíly mezi zmíněnými jazyky. Touto problematikou se zabývá poslední část 3.3. Porovnává jazyky a ukazuje, s jakými problémy se můžeme při vytváření překladače potkat.

3.1 Octave

Octave je software zaměřený na numerické výpočty. Často je využíván na řešení lineárních a nelineárních rovnic, lineární algebry, statistiky a výpočtu matematických experimentů. Může být použit také pro automatické zpracování dávkových souborů. Program dává k dispozici též vykreslování grafů pomocí standardu OpenGL.[7]

Jméno Octave též nese interpretovaný strukturovaný programovací jazyk s vysokou mírou abstrakce. Tento jazyk je jednou z hlavních bezplatných alternativ k jazyku Matlab, ke kterému se blíží svou syntaxí[1]. Nepříjemností pro programátory může být fakt, že jazyk neumožňuje předávat hodnoty do funkcí odkazem. Jazyk je šířen jako svobodný software pod licencí GPL a je hojně využíván vědci a inženýry. [25]

Do verze 3.8 byl Octave pouze dostupný jako CLI[2], ale od této verze získal navíc též GUI[3].[6]

Příklad kódu Použití jazyku Octave můžete vidět na příkladu 3.1.

```
octave1> a = [1 1 1;2 2 2;3 3 3];
octave2> b = [1 2 3;4 5 6;7 8 9];
octave3> a + b
octave4> ans = 2  3  4
octave5>      6  7  8
octave6>     10 11 12
```

Obrázek 3.1: Kód v Octave

[1] Soubor pravidel definující formu jazyka
[2] Command Line Interface - rozhraní příkazového řádku
[3] Graphical User Interface - grafické uživatelské rozhraní

3.2 Eigen

Informace o této knihovně vycházejí z oficiální stránky knihovny [10], která je hlavním zdrojem informací. Je také čerpáno z prezentací, které pořádali autoři knihovny [9] nebo vyučující na univerzitách ve světě [24].

Eigen je knihovna jazyka C++, která se specializuje na lineární algebru, počty s maticemi a vektory, geometrické transformace a též na numerické metody. Tato knihovna spadá pod svobodný software, který je šířen pod licencí MPL2[4].

V knihovně Eigen se využívá metoda šablonového meta-programování, což umožňuje optimalizovat jednotlivé funkce pro různé vstupy [5]. Využití této knihovny nevyžaduje žádnou instalaci či překlad. Funkcionalitu zajišťují hlavičkové soubory, které obsahuji předpis funkcí napsaných v šablonách.

Využití této knihovny lze najít v široké škále oborů, kde jsou potřeba při výpočtu matice. Například v simulacích, hrách, zpracování obrazu a zvuku, robotice, počítačovém vidění atd. Knihovna poskytuje velké množství výhod, některé z nich jsou zmíněny ve výčtu níže.

Výhody

- Mnohostrannost – Díky použití šablon se vyhodnocování matic přizpůsobí jejich velikosti (provede se ideální metoda vyhodnocení). Poskytuje mnoho vestavěných funkcí (FFT, maticové operace, atd.). Podporuje více standardních numerických typů (čísla komplexní, celá a s plovoucí čárkou).
- Rychlost výpočtu – Eigen využívá líné vyhodnocování[6], optimalizace u matic s předem známou velikostí a v některých případech rozvinutí[7] cyklů. V testech výkonnosti v porovnání s ostatními knihovnami si vede v některých odvětvích velmi dobře, v některých průměrně. [21]
- Další výhody – spolehlivost (testována, dokumentovaná a dekomponovaná), známá syntax (využívá konstrukce jazyka C++).

Příklad kódu Na následujícím příkladě 3.2 si můžeme všimnout, jak tato knihovna používá operátor << pro inicializaci matice.

```
Eigen::MatrixXd matrix_a(3,3);
matrix_a << 1,1,1,2,2,2,3,3,3;
Eigen::MatrixXd matrix_b(3,3);
matrix_b << 1,2,3,4,5,6,7,8,9;
std::cout << "Matrix sum:" << std::endl
<< matrix_a + matrix_b << std::endl;
```

```
Output:
Matrix sum:
2  3  4
6  7  8
10 11 12
```

Obrázek 3.2: Kód v Eigenu

[4] MPL - Mozilla Public License - licence, která umožňuje kombinovat tento svobodný software s nesvobodným softwarem (volnější než GPL)

[5] https://eigen.tuxfamily.org/dox/TopicInsideEigenExample.html

[6] Lazy evaluation - hodnota se vyhodnotí až je potřeba k jinému výpočtu

[7] Rozložení příkazů těla cyklu za sebe. Způsobí lepší lokalitu odkazů — příkazy jdou v paměti po sobě

3.3 Porovnání jazyků

Jazyk Octave a knihovna Eigen (v jazyce C++), jsou jazyky zabývající se podobným polem působení. Funkcionalita je bohužel téměř jediná věc, co je spojuje. Octave má širší pole působnosti, umí vizualizovat výsledky (například formou grafů), na rozdíl od Eigenu, který má pouze textový výstup.

Též se liší styl překladu jazyků. Octave je interpretovaný jazyk[8], na rozdíl od C++, který je kompilovaný[9]. Tento rozdíl je velmi markantní a způsobuje mnoho obtížností v implementaci překladače. Například při definici proměnné. V Octavu při přiřazení se nerozlišuje, jestli daná proměnná byla již definována, na rozdíl od C++, kde při druhé definici stejné proměnné překladač zahlásí výskyt chyby.

Dalším důležitým rozdílem je typovost jazyků[10]. Eigen jako C++ knihovna je typovaný[11] a Octave jako většina interpretovaných jazyků je netypovaný[12]. Tato rozdílnost způsobuje problémy s přiřazováním různých typů do proměnné. V Octave dynamicky mění typ dané proměnné na rozdíl od Eigenu, který si hlídá typy svých proměnných. V případě přiřazení jiného typu nastane chyba.

Důležitým rozdílem jsou též rozdílné operátory, které dané jazyky poskytují. Octave rozšiřuje operátory C++ o mocninu, transpozici a sadu operací nad každým prvkem, které pomáhají při výpočtech matic. Rozdílná je také priorita operátorů, která nekoresponduje s C++ prioritou.

Jazyk Octave poskytuje ve funkcích možnost vracet více hodnot, na rozdíl od C++, kde se dá vrátit maximálně jedna hodnota.

Rozdílná je též jejich licence. Octave je vázán licencí GPL, jež neumožňuje využít tento jazyk bez toho, že by celý software nebyl též pod licencí GPL, což znamená zcela volně přístupný. Toto omezení nemusí vyhovovat firmám, které si chrání svoje KNOW-HOW. Pro tyto firmy může být Eigen výhodnou alternativou, jež je zavazuje pouze k zveřejnění části, která vychází z převzaté knihovny.

Shrnutí tohoto porovnání je zobrazeno v tabulce 3.1[13].

	Octave	Eigen
Druh výstupu	Textový a grafický	Textový
Typovost	Netypovaný	Typovaný
Způsob vyhodnocování	Interpretovaný	Překládaný
Návratová hodnota	Více	Jedna

Tabulka 3.1: Tabulka shrnující porovnání jazyků

[8] Jazyk, který se provádí přímo ze zdrojového kódu skrze program interpret, který ho vyhodnocuje v reálném čase

[9] Překládá se do binárního souboru, který pak slouží ke spuštění

[10] Udává, jestli proměnné jsou vázány na datový typ

[11] Proměnná je vázána na datový typ

[12] Datový typ se odvozuje až při zpracování programu

[13] Schéma tabulky převzato z: http://tex.stackexchange.com/questions/67586/how-to-create-comparison-tables-in-latex

Kapitola 4

Návrh překladače

Tato kapitola se zabývá návrhem překladače. V první podkapitole 4.1 jsou napsány důvody, proč bylo rozhodnuto vydat se cestou implementace bez použití existujících nástrojů. Druhá podkapitola 4.2 upřesňuje kompetenci jednotlivých tříd a jejich konečný rozsah v práci. Poslední podkapitola 4.3 zobrazuje základní komunikaci mezi jednotlivými třídami překladače.

4.1 Výběr vhodných nástrojů

Při volbě způsobu implementace překladače existují dvě řešení. Buďto implementovat celý překladač, nebo použít nástroj pro jeho tvorbu. Zvolil jsem první možnost z těchto důvodů:

- Nástroje jsou příliš komplikované pro rozsah práce, který má postihnout pouze podmnožinu jazyka.
- Nástroje vyžadují pokročilé znalosti překladačů a orientaci v nich – mají často velmi komplikovanou strukturu.
- Nástroje vyžadují naučit se jazyku, ve kterém se píší.
- Nástroje často provádějí plno akcí, které nejsou potřebné – zpomalují překlad.
- Méně pokročilé nástroje se nezaměřují na překlad z interpretovaného do kompilovaného jazyka (Octave → Eigen).

Nástroje, z kterých bylo vybíráno pro implementaci, jsou uvedeny a popsány v podkapitole 2.4.

4.2 Rozvržení do tříd

Jednotlivé třídy jsou odvozeny z jednotlivých fází z podkapitoly 2.2, ostatní třídy většinou tvoří datové struktury, které jsou využívány jednotlivými fázemi. Tato podkapitola se zabývá návrhem rozdělení zodpovědnosti v překladači jednotlivým třídám a též zdůvodněním velikosti jednotlivých tříd po implementaci.

Zodpovědnost jednotlivých tříd je rozdělena následovně:

Parser – Zaobaluje celý překladač. Ukládá se v něm celý kontext překladače. Řídí přechody mezi jednotlivými fázemi. Implementuje algoritmus LL syntaktické analýzy shora dolů.

Argument_parser – Zajišťuje čtení a ukládání do souborů. Kontroluje argumenty na vstupu.

Token – Slouží jako datová struktura k uložení přijímaných logických celků s informacemi o nich.

Scanner – Lexikální analyzátor, který zpracovává vstup a plní informacemi objekt Token.

Keywords_table – Shromažďuje informace o klíčových slovech jazyka Octave.

Compiler_exception – Shrnuje výjimky, které mohou nastat v rámci celého programu.

Parser_stack – Modeluje zásobník pro syntaktickou analýzu shora dolů a pravidla pro derivaci neterminálů při ní.

LL_table_rules – Modeluje LL tabulku, v níž jsou uloženy možnosti, kdy se dané pravidlo může derivovat podle vstupního terminálu.

Precedence_analysis – Zajišťuje správnou syntaxi výrazů pomocí precedenční analýzy.

Code_generator – Generuje kód jak pro řídící struktury, tak pro výrazy.

Model_function – Třída držící informace o obsahu funkce.

Symbols_table – Slouží k uchování kontextu používání proměnných a funkcí v programu.

Code_generation_unknown_operations – Nahrazuje neznámé operace za blok kódu se známými operacemi.

Compiler_enumeration – Shromažďuje výčty, které se používají napříč třídami.

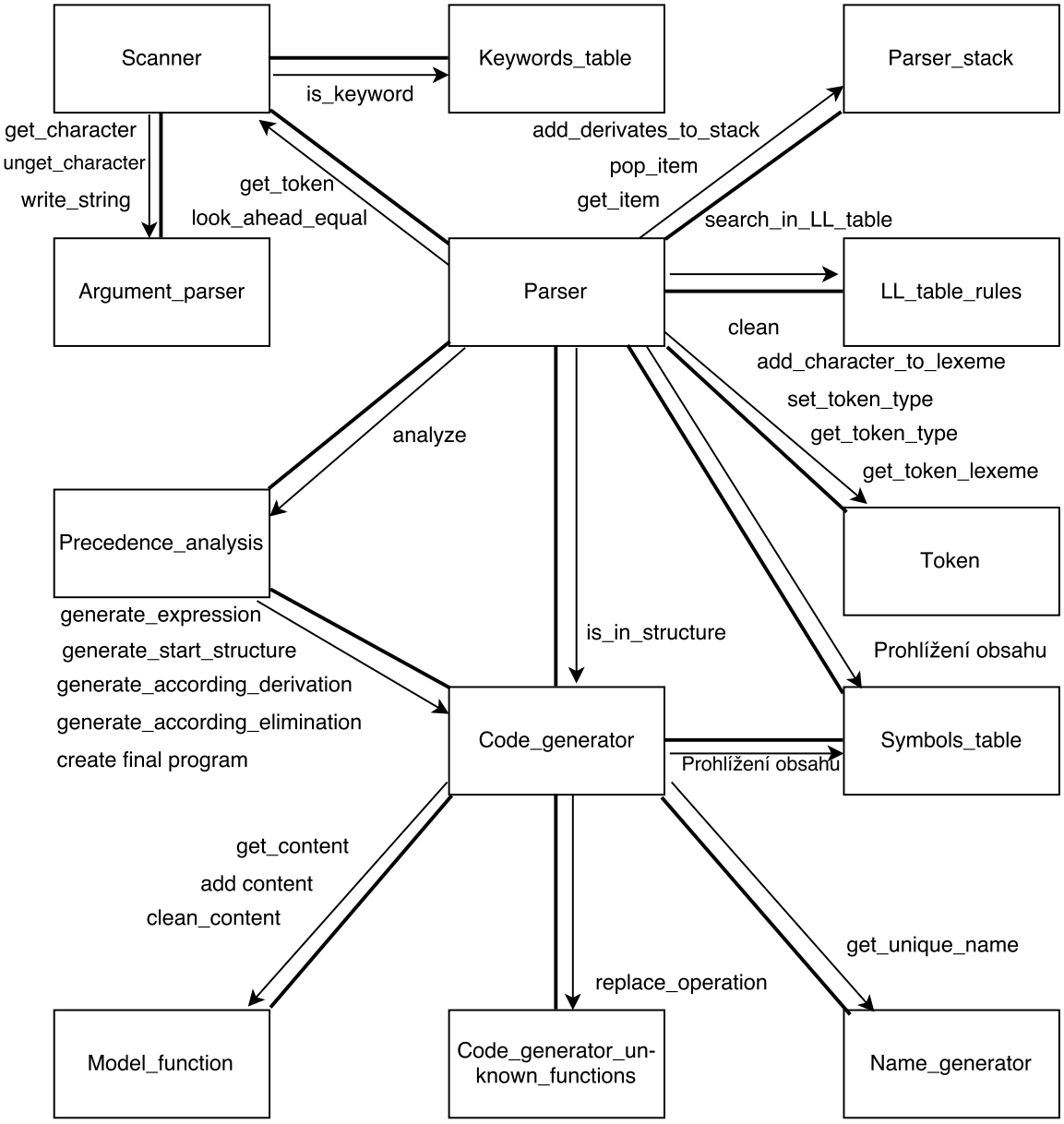
Name_generator – Generuje unikátní názvy pro proměnné.

Po implementaci se ukázala třída `Code_generator` jako mnohem větší než ostatní třídy. Je to způsobeno tím, že zastává stěžejní funkci tohoto překladače, konkrétně kompletním zpracováním sémantiky výrazů.

4.3 Schéma překladače

Překladač je sestaven z několika základních částí, které odpovídají jednotlivým fázím překladu. Na obrázku 4.1 je znázorněn diagram, který znázorňuje komunikaci mezi jednotlivými třídami. V diagramu[1] nejsou pro přehlednost uvedeny třídy `Compiler_exception` a `Compiler_enumeration`, které poskytují podporu většině z tříd pro vyhození specifické výjimky a též poskytují výčtové typy, které se využívají napříč třídami. Metodou prohlížení obsahu je myšleno, že je velké množství metod pro přístup k prvkům této třídy a jejich vypsání by znepřehlednilo diagram.

[1] Vzor převzat z: http://www.agilemodeling.com/artifacts/communicationDiagram.htm



Obrázek 4.1: Komunikační diagram UML

Kapitola 5

Implementace

Kapitola popisuje implementaci jednotlivých částí překladače. V samotné implementaci se využívá anglický jazyk ke komentářům a k pojmenování jednotlivých funkcí a proměnných. V této kapitole se vysvětluje funkcionalita jednotlivých funkcí a provázanost jednotlivých fází a modulů.

První fází implementace je zpracování vstupních argumentů 5.1, které poskytuje komunikaci s uživatelem. Druhá implementační fáze, lexikální analýza 5.2, se zabývá způsobem zpracování vstupního souboru a přípravě vhodné reprezentace pro samotný překlad.

Ve třetí fázi syntaktické analýzy 5.3 se odehrává celková analýza vstupního programu. Čtvrtá fáze, sémantická analýza 5.4, sekunduje třetí a pomáhá odchytit některé chybné zápisy přijímaného jazyka.

Poslední z fází je generování cílového kódu 5.5, které vytváří výstup v jazyku C++ konkrétně knihovně Eigen. Poslední část této kapitoly se zabývá nejdůležitější datovou strukturou překladače, a to tabulkou symbolů 5.6, která napomáhá s analýzou a syntézou kódu.

5.1 Zpracování vstupních parametrů

Konstruktor třídy `Argument_parser` kontroluje počet argumentů a pokouší se otevřít soubory pro čtení a zápis. Když nastane chyba, způsobí uzavření všech již otevřených souborů a vyvolání výjimky, kterou odchytí volající strana. Streamy[1] otevřených souborů jsou uložené jako členské proměnné.

Streamy jsou zapouzdřeny ve třídě přístupné pouze pomocí funkcí `get_character` a `unget_character`, které poskytují pro lexikální analýzu překladače přečtení a navrácení znaku ze souboru. Dále je zde funkce `write_string`, která umožňuje zapsat výstupní kód do výstupního souboru. Při destrukci objektu se uzavírají všechny otevřené soubory.

5.2 Lexikální analýza

Lexikální analýza je implementována pomocí deterministického konečného automatu ve třídě `Scanner`. První metodou, která komunikuje se syntaktickou analýzou, je metoda `get_token`, kde syntaktická analýza předává v argumentu objekt třídy `Token`, do kterého nahrává lexikální analýza informace o lexému. `look_ahead_equal` je druhou veřejnou metodou, která kontroluje následující znak (kromě mezer), jestli se rovná znaku rovná se.

[1] Stream = Textový proud

Objekt třídy `Token` poskytuje prostor pro uchování datového typu zpracovaného lexému. Pro přístup k jednotlivým členským proměnným slouží metody `get_token_lexeme` pro přístup k jménu proměnné a `get_token_type` ke zjištění jejího typu. Třída dále poskytuje též nastavení těchto hodnot pomocí metod `set_token_type` a `add_character_to_lexeme`, která přidává na konec řetězce předávaný znak. Pro vymazání obsahu objektu je zde metoda `clean`.

Jednotlivý stav automatu představuje typ tokenu, který je nyní vybrán pro daný lexém. Mezi jednotlivými typy přechází podle určených přechodových pravidel v automatu. Typ tokenů je ustálen až po zjištění začátku dalšího tokenu. Po zvolení typu se token odešle a předá se tok programu syntaktické analýze.

Ve stavu zpracování identifikátorů se analyzátor rozhoduje, jestli je daný řetězec identifikátor, nebo klíčové slovo. Ke zjištění příslušnosti daného tokenu využívá tabulky klíčových slov, kterou poskytuje instance třídy `Keyword_table`, umístěná v privátních proměnných lexikálního analyzátoru. Po zjištění konce lexému se analyzátor podívá do tabulky pomocí metody `is_keyword`, jestli daný identifikátor je klíčové slovo. V případě pozitivního nálezu se změní typ daného tokenu.

Porovnání s větším množstvím jednotlivých znaků je seskupeno do privátních metod, které zpřehledňují určování typů v automatu.

5.3 Syntaktická analýza

Syntaktický analyzátor je základním kamenem programu. Je implementován pomocí třídy `Parser`, která v sobě vytváří všechny potřebné instance tříd, které zajišťují tyto fáze překladu: syntaktickou, sémantickou analýzu a generování cílového kódu.

Implementace v sobě zahrnuje i začátek celého programu – funkci `main`, kde se otvírají potřebné soubory konstrukcí instance třídy `Argument_parser`. Poté se vytváří instance analyzátoru, která po spuštění metodou `analyze` provede překlad celého souboru na vstupu překladače. Během překladu může překladač vyvolat výjimku, která je způsobena špatným uživatelským vstupem, omezením překladače nebo vnitřní chybou programu. Když je výjimka zachycena, vypíše se informace, v jaké fázi nastala a jaká je její příčina. Definice druhů vlastních výjimek je centralizovaně umístěna do hlavičkového souboru `Compiler_exception.h`.

5.3.1 Analýza založena na LL tabulce

Syntaktická analýza začíná zavoláním metody `get_terminal`, která požádá lexikální analýzu o token. Pokud v tokenu je umístěná operace, která není implementována ve stávající verzi programu, vyvolá se informující výjimka. Následně tento token je ověřen, jestli má být transformován (hlouběji analyzován) metodou `is_terminal_needs_translate`. Za provedení transformace zodpovídá metoda `translate_to_terminal`, která specifikuje identifikátor a převádí do vnitřního formátu konstantu nebo ukončující znak. Pokud metoda zjišťuje typ v tabulce symbolů, používá k tomu metodu `translate_to_parser_annotation`, která překládá typy proměnných na typy identifikátorů. Vstup je nyní již ve tvaru, který je potřebný pro zahájení syntaktické analýzy.

Při instanciaci objektu třídy `Parser` se konstruují všechny objekty potřebné k analýze programu. Jednotlivé třídy objektů budou postupně během implementace představeny. Implementaci tříd lze nalézt na místech:

- O třídách `Scanner` a `Token` je pojednáno v podkapitole 5.2 o lexikální analýze.

- Třídou `Precedence_analysis` se zabývá implementace precedenční analýzy v podpodkapitole 5.3.2.
- Třídě `Code_generator` je věnována celá podkapitola 5.5 o generování cílového kódu.
- O třídě `Symbols_table` pojednává podkapitola 5.6, která zastupuje tabulku symbolů.
- Třídy `LL_table_rules` a `Parser_stack` slouží k prediktivní analýze, kterou se zabývá tato sekce, proto jsou popsány na následujících řádcích.

`LL_table_rules` je třída, která při instanciaci si naplní hašovací tabulku `rule_LL_tab` informacemi o tom, jaké pravidlo pro derivaci se má použít při příchodu daného terminálu, když na zásobníku `stack` objektu třídy `Parser_stack` je daný neterminál. Ke zjištění toho pravidla slouží jediná veřejná metoda `search_in_LL_table`, která přijímá jako argumenty neterminál a terminál.

Grafické znázornění LL-tabulky, ve které se vyhledává následující pravidlo, je v příloze C. Pro velký počet možných terminálů je tato tabulka rozdělena do šesti částí. Záznamy z této tabulky vycházejí z množiny Predict, která byla odvozena z pravidel v příloze B, kde je s nimi též zapsána.

Třída `Parser_stack` vytváří zásobník `stack` pro syntaktickou analýzu shora dolů. Do tohoto zásobníku jsou ukládány symboly (terminály, neterminály). Metody této třídy dovolují nahlížet na vrchol zásobníku `get_item`, smazat symbol na vrcholu zásobníku `pop_item` nebo derivovat neterminál na vrcholu zásobníku `add_derivates_to_stack`, jehož argumentem je číslo pravidla rozpoznaného v LL tabulce. Pravidla pro derivaci jednotlivých neterminálů podle LL tabulky jsou uloženy v poli `derivates`. Jednotlivá pravidla pro derivaci jsou zaznamenána v příloze B.

Sám průběh prediktivní analýzy probíhá v nekonečném analytickém cyklu, který může skončit pouze úspěšným překladem, nebo chybou. Analyzátor se rozhoduje podle symbolu na vstupu a podle vrchního symbolu na zásobníku. Ke zjištění typu vrchního symbolu na zásobníku pomáhají metody `is_expression` a `is_nonterminal`. Metoda využívá princip zásobníkového automatu viz. definice 2.1.2.2. Nekonečný cyklus prediktivní analýzy je zobrazen v algoritmu 1.

5.3.2 Precedenční analýza výrazů

Při instanciaci objektu třídy `Precedence_analysis` precedenční analýzy se v konstruktoru naplní tabulka prioritami operátorů[2], které jsou uložené v `precedence_table`, dvojrozměrném poli, které je graficky znázorněno v příloze D. Priorita a asociativita jednotlivých operátorů je sestavena podle skupin priorit operátorů z tabulky 5.1.

Též se vytvoří seznam pravidel redukcí `reduction_rules`, které slouží ke kontrole syntaktické správnosti výrazu. Předpis redukčních pravidel je zaznamenán v příloze E.

[2] přednost operátorů při vyhodnocování

```
while (True) do
    if neterminal then
        if rule != ll_table_search_rule() then
            exit error;
        else
            stack.pop();
            stack.add_derivates_to_stack()
        end
    else if expression then
        precedence_analysis.analyze();
        stack.pop();
        read_terminal;
    else
        if terminal != stack.pop() then
            exit error;
        else
            stack.pop();
            if terminal == EOF then
                if stack.top()==$ then
                    exit success;
                else
                    exit error;
                end
            end
        end
    end
    read_terminal;
end
```

Algoritmus 1: Hlavní cyklus prediktivní analýzy

Priorita	Operátory	Asociativita
1	‘()’	zleva
2	postfixové ‘++’ ‘- -’	zprava
3	‘^’ ‘.^’ ‘**’ ‘.^’ ‘.**’	zleva
4	‘+’ ‘-’ ‘!’ prefixové ‘++’ ‘- -’	zleva
5	‘*’ ‘/’ ‘\’ ‘.\’ ‘.*’ ‘./’	zleva
6	‘+’ ‘-’ ‘.+’ ‘.-’	zleva
7	‘<’ ‘<=’ ‘==’ ‘>=’ ‘>’ ‘!=’	zleva
8	‘&’	zleva
9	‘|’	zleva
10	‘&&’	zleva
11	‘||’	zleva
12	‘=’ ‘+=’ ‘-=’ ‘*=’ ‘/=’ ‘\=’ ‘.*=’ ‘./=’ ‘.\=’ ‘.^=’, ’**=’, ’^=’, ’.**=’, ’.^=’ ‘.+=’ ‘.-=’	zprava

Tabulka 5.1: Tabulka precedence operátorů

Analýza přijímá první token od prediktivní analýzy. Tento token si převede pomocí metody `classified` do své interní reprezentace neterminálů a terminálů. Poté, když potřebuje k další analýze nový token, volá metodu `get_new_classified_token`, která jí vrátí nový token už převedený do vnitřní reprezentace.

Na začátku analyzačního cyklu se ukládá token do pole `expression`, které posléze slouží jako zdroj informací pro generování cílového kódu.

Pokud se rozpozná v klasifikaci, že proměnná je funkce (pomocí náhledu do tabulky funkcí), zkontroluje se existence argumentů ohraničených v kulatých závorkách. V závorkách se zpracují argumenty pomocí metody `read_call_func_arguments`. Po přečtení celého volání metody se pokračuje dále v analýze.

Sama precedenční analýza se rozhoduje mezi třemi možnostmi, a to redukcí anebo nahrání symbolu na zásobník, buďto s vložením závorky před terminál, nebo bez vložení závorky. Daná možnost je vybrána na základě priority vstupního symbolu a symbolu na vrchu zásobníku zpřístupněného metodou `get_top_terminal`. Analýza skončí úspěšně, pokud pomocí redukcí výrazu dojde na počáteční neterminál. Pokud se nenajde pravidlo pro redukci, nastane chyba, vyvolá výjimku s informacemi pomáhající uživateli lokalizovat chybu. Algoritmus 2 ukazuje v pseudokódu fungování precedenčního analyzátoru.

```
repeat
    switch precedence_table[stack.top()][actual_input] do
        case smaller do
            push_brace_behind_terminal();
            stack.push(actual_input);
            get_token();
        case greater do
            reduction();
        case equal do
            stack.push(actual_input);
            get_token();
        otherwise do
            exit error;
        end
    end
until actual_input == $ && stack.top(terminal) == $;
```

Algoritmus 2: Hlavní cyklus precedenční analýzy

K nalezení vhodného pravidla pro redukci výrazu se používá metoda `search_rule`, která vyhledává vhodné pravidlo v tabulce pravidel `reduction_rules`. Pro lepší přehlednost přístupu k redukčním pravidlům je zavedena enumerace `reduc_rule`.

Zaobalení pro přidávání prvků na zásobník precedenční analýzy zastávají metody `push_to_stack` pro vložení prvku na zásobník a `push_brace_behind_terminal` pro přidání závorky před poslední terminál.

Během průběhu analýzy se počítají závorky kvůli vrácení kontextu prediktivní analýze nalezením konce výrazu. Pokud je nadbytečná uzavírající závorka, předává se řízení programu zpět hlavní LL analýze i s touto závorkou.

Po ověření syntaktické správnosti výrazu se předává výraz generátoru cílového kódu 5.5.

Na konci celé analýzy se uvede instance do původního stavu metodou `clean_analysis`.

5.4 Sémantická analýza

Sémantická analýza je implementací úzce spjata s generováním kódu tím, že probíhá uvnitř této fáze. I když překladač předpokládá validní vstup, přesto pro správný vnitřní chod programu se kontroluje u některých prvků sémantická správnost. Analýza má charakter osamocených sémantických akcí více než souvislého bloku analýzy.

Jedním z úkolů sémantické analýzy je kontrola správných výstupních typů. Problém rozdílu dělení v jazyku C++ (implicitní celočíselné) a Octave (implicitní neceločíselné) je řešen v operacích dělení pomocí explicitního přetypování jednoho z operandu na neceločíselné číslo. S tímto problémem se též vypořádává analýza typů, která počítá s nutností přiřazení výsledku do neceločíselné proměnné.

Další funkce sémantické analýzy je kontrola části vstupního programu v jeho kontextu. Zabývá se kontrolou typů operandů při generování operací. Například kontrolování správných typů operandů při mocnině matice. Stará se též o kontrolu správného zápisu matic. Pokud narazí na nesrovnalost, hlásí ji uživateli pomocí výpisu na standardní chybový výstup.

Sémantická analýza má též na starost zjištění správného typu vznikající proměnné. Tato část je popsána v podpodkapitole 5.5.2 v odstavci Přiřazení do proměnné, kde poskytuje výsledný typ výrazů pro generování přiřazení.

5.5 Generování cílového kódu

Řízení programu je vždy předáno generování cílového kódu, probíhající v objektu třídy `Code_generate`, syntaktickou analýzou po zpracování terminálu, při derivaci neterminálu viz v podpodkapitole 5.5.1, nebo výrazu ze syntaktické analýzy viz v podpodkapitole 5.5.2.

Kromě hlavních metod generování cílového kódu popsaných v podpodkapitolách jsou k dispozici další metody, které poskytují výstupy z generování. Jedna z těchto metod `generate_start_structure()` poskytuje inicializaci výstupního programu vygenerováním hlavičky (informace o překladači a potřebných knihoven) kódu. Informaci, zda-li se nacházíme uprostřed zvolené řídící struktury, poskytuje metoda `is_in_structure`. Poslední metoda `create_final_program` sesbírá data z celého generátoru a vrátí celý kód výsledného programu.

5.5.1 Generování struktury kódu

První část generování struktury se děje v `generate_according_derivation`, kde se generují hlavičky řídících struktur podle použitého LL pravidla. Při zanořování do jednotlivých struktur zde vznikají nové stupně tabulky symbolů a u zanoření do funkcí celá nová tabulka symbolů. Funkce též způsobí, že ukládání generovaných výrazů je přesměrováno z hlavní funkce na právě definovanou funkci.

Generování na základě pravidel potřebuje pro zjištění svého chování vědět, v jaké řídící struktuře se vyskytuje. K tomu mu slouží metoda `number_control_flow`, která vrací pravidlo určující řídící strukturu, ve které se nachází.

Metoda `create_for_cycle_matrix_head` vytváří cykly pro iterovaní nad maticemi v řídící struktuře for cyklu. Informaci pro konstrukce break a continue zjišťuje metoda `is_in_cycle`, zda-li jsou tyto příkazy v cyklu. Tato kontrola je zde hlavně kvůli tomu, že při použití těchto konstrukcí mimo cyklus v jazyku Octave nezpůsobí nic, na rozdíl od jazyka C++, kde způsobí selhání překladu.

Inicializace parametrů funkce probíhá v metodě `copy_parameter`, kde se po předání argumenty zkopírují a přiřadí se do hodnot parametrů. Tento krok je nutný kvůli způsobu předefinování proměnných v překladači.

Druhá část, o kterou se stará metoda `generate_according_elimination`, uzavírá každou řídící strukturu specifickým způsobem. S vygenerováním specifického ukončení se též vymaže informace o tom, v jaké řídící struktuře se nacházel generátor. K tomuto účelu slouží metoda `clean_control_flow`, které pomáhá metoda `type_of_control_flow`, která hledá zarážku v podobě pravidla řídící struktury. V případě ukončení funkce ji přidá do tabulky `Generic_functions`.

Stará se též o rozdílnost ukončovací podmínky cyklu typu do-while, kde na rozdíl od C++ iterace přes cyklus končí tehdy, až ukončovací podmínka je nesplněna. U jazyka Octave platí, že se vykonává, dokud se podmínka nesplní.

5.5.2 Generování výrazů

Tato podpodkapitola neřeší podrobně každou metodu, ale zabývá se hlavními bloky generování kódu, které mají na starosti jednotlivé problémy. Generování výrazu se začíná v metodě `generate_expression`, která se stará o sestavení výrazu z vlastní definice a deklarace. Pro naplnění těchto částí se volá hlavní metoda v generování výrazů `expression_process`. Tato metoda je komplexnější, proto jednotlivé části jsou blíže popsány na následujících řádcích.

Vyhledání operace rovná se Důležitým bodem vyhodnocení výrazu je zjistit, zda-li se v něm vyskytuje přiřazení. Zjištění, jestli se vyskytuje rovná se a na jaké konkrétní pozici poskytuje metoda `process_expression_assign`. Pokud se nenajde rovná se, neprovádí se část generování výrazu Přiřazení do proměnné.

Nahrazení konstant a předefinovaných proměnných Nahrazení konstant v kódu zabezpečuje metoda `transform_constants`, která hledá ve výrazu konstanty jazyka Octave, které se zamění na ekvivalentní konstanty v jazyku C++. Odpovídající hodnoty se hledají v tabulce `constants_table`. Též tato metoda poskytuje převod ze zápisu matic v jazyku Octave do vnitřní reprezentace. Pro lepší práci se ukládá do nové proměnné, která je získána z objektu generátoru jmen metodou `get_unique_ident`. Tento generátor si pamatuje svůj vnitřní stav čítače a vytváří proměnné s prefixem `_oct_to_cpp_` a zakončené číslem čítače.

Kvůli možnosti redefinice proměnných se vyhledává, jestli daná proměnná již nebyla předefinována. Ke kontrole změny jména proměnných ve výrazu slouží metoda `rename_all_redefined_variables`. Metoda projíždí celý výraz a hledá jména proměnných, které se pozměnili změnou datového typu proměnné. Pokud takovou proměnnou najde, nahradí jí novým jménem.

Přiřazení do proměnné O celý proces přiřazení výrazu do proměnné se stará metoda `assignment_variable`. Tato metoda se skládá z níže popsaných pomocných metod.

V metodě `type_of_assignment_variable` se zjišťuje, jaký datový typ má výraz za přiřazovacím operátorem. Metoda provádí určení datového typu na základě priorit operátorů, projde se přes celý výraz a pomocí metody `find_out_binary_operator` se najde operace s nejvyšší prioritou. Podle této operace se rozdělí celý výraz na dva podvýrazy. Tyto podvýrazy jsou znovu analyzovány. Takto se postupuje, dokud v podvýrazu není osamocená proměnná, u niž již není problém určit její datový typ z tabulky symbolů. Též se bere

v potaz při určování výsledného typu, že některé z operací mohou vrátit rozdílný typ než jsou operandy.

Kvůli většímu množství typu přiřazení se rozlišuje přiřazení s rovná se od ostatních přiřazení. Ostatní přiřazení se váží vždy s některou z operací, kde výsledný typ je určen stávajícím typem a typem přiřazení v metodě `type_of_expression_according_to_operation`.

Uložení proměnné, do které se přiřazovalo, do tabulky symbolů má na starosti metoda `assign_variable_to_symbols_table`. Tato metoda též hlídá, jestli daná proměnná již byla definovaná. Pokud byla již definována pod jiným typem, tak se vytvoří nová proměnná, která od této chvíle bude zastupovat původní proměnnou.

Následně se kontroluje, jestli proměnná, do které bylo přiřazováno, nebyla předefinovaná. Když byla v tomto výrazu předefinovaná, změní se její jméno na současné.

Vyhodnocování funkcí O vyhodnocení funkcí se stará metoda `call_function`. Zjišťuje, jestli daná volaná funkce s danými datovými typy parametru je již definovaná. Pokud není, zjistí, jestli funkce je vestavěná. Pokud funkce není vestavěná, tak se podívá, jestli není již mezi typově nespecifikovanými (generic) funkcemi. Když ji zde nalezne, vytvoří se nová specifická funkce a přidá se do funkcí `Functions`. Pokud funkce s daným jménem neexistuje, tak nebyla definovaná a vypíše se chyba. Výsledek dané funkce se přiřadí do nové proměnné, která je pak nahrána do návratové hodnoty funkce.

Kvůli stejným názvům některých vestavěných funkcí Octave a C++ se musí vestavěné funkce definovat a deklarovat ve vlastním jmenném prostoru `OctGen_function`. Proto se explicitně nastavuje u vestavěných funkcí jmenný prostor překladače.

Transformace neznámých operací O překlad neznámých operací se stará metoda `operation_transform`, která provádí metodu `modified_operation` nad každou nedefinovanou operací uloženou v seznamu `list_of_transform_operation`. Tato metoda rozdělí výraz na dvě části podle neznámé operace s nejvyšší prioritou. Nad každou z rozdělených částí se spustí hledání neznámé operace. Když se zanoření dostane na výrazy, které v sobě již nemají žádné neznámé operace, začnou se vynořovat z rekurze. Dané analyzované části (podvýrazy) předáváme jako operandy metodě `replace_operation`. Metoda vypočte v deklaraci výrazu danou neznámou operaci a uloží výsledek do nové proměnné, se kterou nahradí daný podvýraz. Metoda vybírá ze skupin neznámých operací způsob, jakým mají být generovány. Tato metoda se svými pomocnými metodami je vyčleněna do modulu `Code_generate_unknown_operations.cpp` kvůli přehlednosti kódu v třídě generování cílového kódu.

Skupiny transformací operací se dělí podle druhu operandů, jestli jsou skalární, nebo jsou to matice, popřípadě jejich kombinace. Při operaci nad dvěma maticemi se musí počítat, že pro některé operace se musejí rozpoznávat operandy matic a vektorů[3].

Každá operace jazyku Octave je v nějakém aspektu trochu rozdílná od operace C++ knihovny Eigen, proto se musí daná operace považovat za neznámou. Jedním z příkladů je dělení, u kterého k docílení neceločíselného dělení (v Octave) je využito explicitní přetypování jednoho z operandů. Výsledky jednotlivých operací jsou ovlivněny druhy operandů. Rozlišují se skaláry, vektory a matice.

Konečná úprava výrazu Metoda `postprocess_replaces_operation` upravuje výraz do výsledné podoby zpracováním unárních operací a též přidáním indexu průchodu cyklu

[3] Jedno dimenzionální matice.

for k proměnné. Též tato část kontroluje, zda-li je ve výrazu jen jedna proměnná. Tuto informaci předává dál na výpis proměnné.

Výpis proměnných Pokud nebyl syntaktickou analýzou ve výrazu rozpoznán středník, vytvoří se konstrukce na vypsání daného výrazu. Pokud je ve výrazu pouze jedna proměnná, výpis je uvozen názvem původní (ne překladačem vytvořené) proměnné a rovná se. Jiné výrazy se uvozují obecným předpisem `ans =`.

5.6 Tabulka symbolů

Tabulka symbolů, implementována třídou `Symbols_table`, je tvořena více datovými strukturami (tabulkami) shromažďující informace o analyzovaném kódu a metodami zpřístupňující informace z nich. V jednotlivých podkapitolách budou probrány jednotlivé tabulky.

Podkapitola začíná tabulkou klíčových slov 5.6.1, která pomáhá s klasifikací identifikátorů. Následuje tabulka konstant 5.6.2 umožňující převádět konstanty z jazyka Octave do jazyka C++. Hlavní datovou strukturou tabulky symbolů je tabulka proměnných 5.6.3, která poskytuje typ uložené proměnné, popřípadě odkaz na její nové jméno. Této tabulce pomáhá s předefinovanými proměnnými tabulka substitucí 5.6.4. Poslední část se zabývá tabulkami funkcí 5.6.5, které se starají o pamatování definovaných funkcí a též interpretací vestavěných funkcí.

5.6.1 Tabulka klíčových slov

Tabulka klíčových slov slouží k rozeznání klíčových slov od ostatních identifikátorů. Při vytvoření se tabulka v konstruktoru inicializuje klíčovými slovy jazyka Octave. Tyto klíčová slova se uloží do hašovací tabulky. Jedinou veřejnou metodou je `is_keyword`, která rozpoznává, zda daný identifikátor je klíčové slovo.

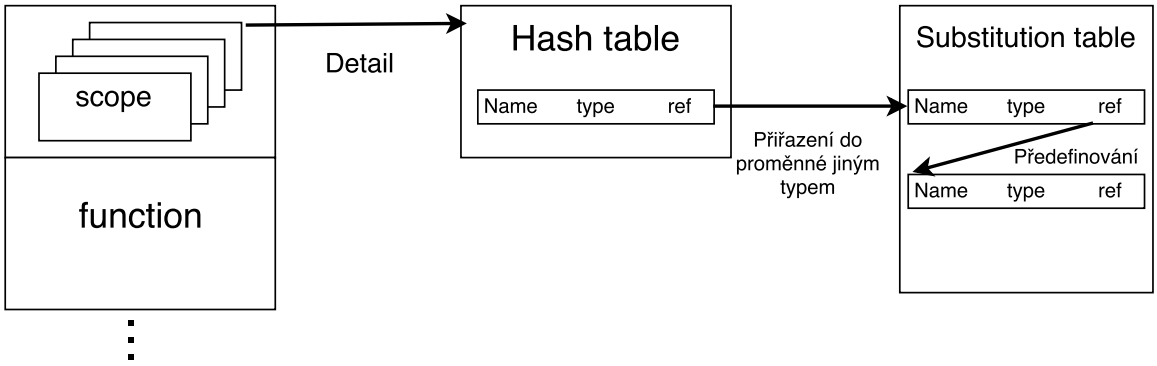
5.6.2 Tabulka konstant

Tabulka konstant poskytuje zjištění, zda daný identifikátor je konstanta. Tato datová struktura nepředpokládá velký počet položek, proto je uložena v poli `constants_table`. Ke zjištění hodnoty konstanty slouží metoda `get_value_constant`, která vrací ekvivalentní konstantu v C++. Též existuje reverzní metoda `get_original_value_constant` pro zjištění původní konstanty Octave.

5.6.3 Tabulka proměnných

Tabulka proměnných `symbols_tables` je datová struktura, která je složena z tabulek symbolů. Tyto tabulky symbolů jsou složeny z rozsahu platnosti (angl. scope) jednotlivých proměnných pro zanořování řídících struktur. Tyto rozsahy platnosti jsou implementovány pomocí hašovacích tabulek, kde se ukládají jednotlivé záznamy proměnných. Pokud je proměnná předefinovaná, vznikne odkaz na prvek v tabulce substitucí, který tuto proměnnou nahrazuje. Princip tabulky proměnných je zobrazen na obrázku 5.1.

K manipulaci s daty v tabulkách slouží veřejné metody. Jsou zde metody pro přidání celých tabulek symbolů `add_symbols_table` pro přidání a `delete_symbols_table` pro odebrání tabulky. Dále též metody pro přidání `add_new_scope` a odebrání `delete_scope` rozsahů platnosti proměnných. Pro přidání jednotlivé proměnné slouží metoda `add_symbol`.



Obrázek 5.1: Struktura tabulky proměnných a její napojení na tabulku substitucí

Vyhledání zajišťuje metoda `find_symbol`, která zjišťuje jestli se daná proměnná vyskytuje v tabulce proměnných nebo v tabulce substitucí.

Pro vyhledávání jednotlivých informací v tabulce proměnných slouží řada metod začínající anglickým slovesem get. Pro získání typu proměnné `get_actual_symbol_type`, pro získání reference konkrétní proměnné `get_actual_symbol_reference`. Je zde také možnost získat typ poslední redefinice dané proměnné pomocí metody `get_symbol_type`. Metoda `get_symbol_type_in_previous_symbols_table` poskytuje zjištění typu dané proměnné z předešlé tabulky symbolů, což umožňuje zjistit typ předávaného argumentu. Též můžeme pomocí metody `find_symbol_in_previous_symbols_table` zjistit existenci proměnné v předešlé tabulce symbolů. K získání prvotního názvu proměnné je určena metoda `get_original_name`. Pro získání informace, zda-li se nacházíme v tabulce symbolů hlavní funkce, slouží metoda `is_main_symbols_table`.

5.6.4 Tabulka substitucí

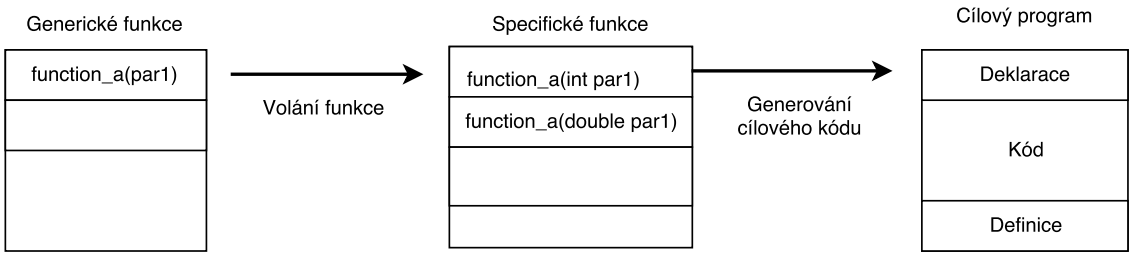
Tabulka poskytuje řešení problému přiřazení do proměnné jednoho typu typem jiným. V C++ tato syntaktická konstrukce vyvolává chybu při překladu, že daná proměnná může být definovaná jen jednou na jeden typ a po zbytek své doby platnosti nemůže změnit typ.

Tabulka tento problém řeší pomocí vygenerování nové unikátní proměnné, kterou vloží do tabulky substitucí pomocí metody `add_to_substitution_table`. K úpravě odkazu se používá metoda `add_to_reference_to_variable`, která ho změní na novou proměnnou. Od této chvíle se používá jméno nové proměnné namísto předefinované. Pokud se vícekrát předefinovává proměnná na jiné typy, odkazy se nastavují jen mezi prvky v tabulce substitucí.

5.6.5 Tabulky funkcí

Tabulky funkcí jsou složeny z tabulky specifických funkcí, tabulky generických funkcí a tabulky vestavěných funkcí.

Při definici uživatelské funkce se pomocí metody `add_generic_function` vytvoří generická funkce, ze které se pak při volání vytváří specifická funkce. Proces definování uživatelských funkcí je znázorněn na obrázku 5.2.



Obrázek 5.2: Proces definování uživatelských funkcí

Při volání funkce se překladač dívá do tabulky specifických funkcí `functions`, jestli již tato definice neexistuje metodou `is_functions_equal`. Pokud existuje, tak se již funkce znovu nezapisuje do tabulky. Pokud neexistuje, tak se zkontroluje, zda není daná funkce v tabulce `build_in_functions` s vestavěnými funkcemi metodou `is_build_in_functions_equal`. Když se zde funkce najde, zkopíruje se daná funkce s určenými typy parametrů do specifických funkcí pomocí metody `load_function_from_build_in_functions`. Pokud se ani zde nenajde, tak se podívá metodou `is_generic_function`, jestli se nachází v generických funkcích. Pokud se ani, zde nenalezne vypíše se, že funkce nebyla definována. Pokud nalezne generickou funkci, zjistí typy argumentů a podle nich si vytvoří parametry s typy pomocí metody `create_parameters`. Tento předpis poté vloží do tabulky se specifickými funkcemi pomocí metody `add_specific_function`.

K převodům mezi jednotlivými reprezentacemi proměnných slouží skupina privátních metod konvertorů.

Na konci celého generování se z tabulky `functions` získají deklarace a definice jednotlivých funkcí. K získání všech deklarací slouží metoda `get_all_declarations` a k získání všech definicí slouží metoda `get_all_definitions`. Tyto metody používají metody implementující "gettery"jednotlivých složek funkce.

5.7 Testovací sada

Testovací sada se skládá z 25 testů, které testují převod základních konstrukcí jazyka Octave do C++ s využitím knihovny Eigen. Testování překladače se spouští z hlavního adresáře pomocí souboru Makefile. Tento makefile spouští ve složce tests delegovaný druhý makefile určený pro testování, který provádí následující operace:

- Na začátku se provedou postupně všechny testovací programy psané v Octave a výsledky se uloží do souborů.
- Následně se spouští transpiler se všemi testovacími Octave zdrojovými kódy. Tímto procesem se vytvářejí přeložené soubory do C++ s využitím knihovny Eigen.
- Poté se tyto soubory přeloží pomocí překladače g++, spustí se a výsledky programu se uloží do souborů.
- Po zachycení výsledků z obou druhů jazyků se spustí skript test.sh, který provádí porovnání každého výstupu z jednoho jazyka s tím druhým pomocí nástroje Diff.
- Výsledný počet úspěšných a neúspěšných testů se vytiskne na standardní výstup a též do souboru test_results.out.

Testové soubory mají příponu .m, které slouží jako zdroj pro generování dalších souborů s následující významem:

- *.m_result.out - Zaznamenaný výsledek provedení Octave skriptu.
- *.cpp - Vygenerovaný kód překladačem OctGen.
- *.cpp.bin - Spustitelný soubor programu z vygenerovaného cpp souboru.
- *.cpp_result.out - Zaznamenaný výsledek spuštění programu vygenerovaném v C++.

Testy se provádí podle lexikografického řazení. To znamená, že je nutné první desítku předznamenat nulou, aby bylo zachované očekávané pořadí testů (vzestupné).

Kapitola 6

Uživatelská příručka

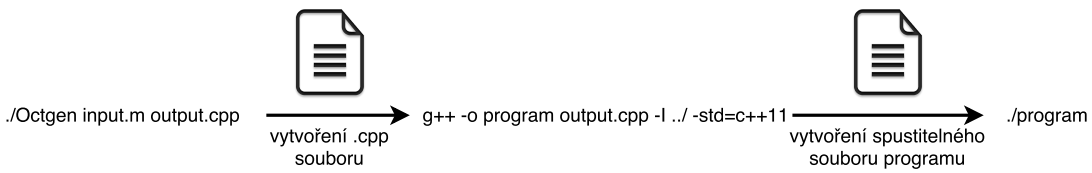
Tato kapitola poskytuje bližší informace k použití překladače. V první části 6.1 je ukázáno, jak překladač nainstalovat a připravit ho k používání. Druhá podkapitola 6.2 ukazuje způsob užití na konkrétním příkladě. Třetí podkapitola 6.3 poukazuje na nutné podmínky, které musejí být splněny při používání překladače. Čtvrtá část 6.4 se zabývá testováním překladače. Ukazuje způsob, jak ověřit správnou funkčnost transpileru. Poslední část 6.5 představuje důvody, proč používat vyvinutý transpiler.

6.1 Instalace

Instalace se provádí příkazem `make` v kořenovém adresáři projektu. Prvním krokem instalace je zjištění, jestli existuje v adresáři knihovna Eigen. Pokud neexistuje, začne se klonovat z repozitáře verzovacího nástroje Git. Stahování může zabrat delší dobu, protože knihovna Eigen je datově objemnější (13 MB). Po dokončení stahování knihovny se spustí samotné sestavení programu, které vytvoří spustitelný binární soubor OctGen ve složce bin.

6.2 Způsob užití

Po instalaci programu můžeme využívat binárního souboru OctGen ve složce bin. Program přijímá dva parametry. První je název(cesta) ke vstupnímu programu a druhý je název(cesta) výstupního programu. Na obrázku 6.1 je ukázaný proces od souboru v Octave až po konečný binární soubor. Celý proces začíná přeložením octave souboru pomocí transpileru do souboru se zdrojovým kódem v C++. Tento soubor se nyní přeloží překladačem C++ a vznikne již konečný binární soubor. Binární soubor je na konci ukázky spuštěn. Při překladu z C++ kódu je doporučeno použít optimalizaci pro zrychlení běhu cílového spustitelného souboru.



Obrázek 6.1: Případ užití programu OctGen.

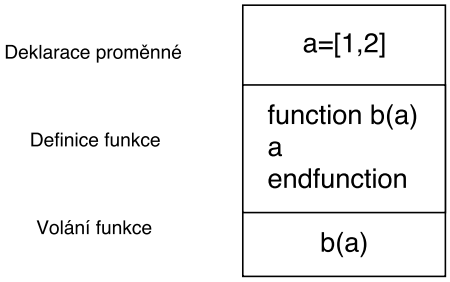
6.3 Nutné podmínky při použití překladače

Překladač kvůli složitosti převodu z netypovaného na typovaný jazyk vyžaduje dodržování určitých níže zmíněných konvencí psaní zdrojového kódu. Uživatel je vždy s omezením překladače seznámen při překladu buď neúspěšným překladem s vypsáním, co je v nepořádku, nebo varovným výpisem na standardním chybovém výstupu.

Funkce První omezení zabraňuje použití výrazů a volání dalších funkcí v argumentech funkcí. Toto omezení prodlužuje pouze zápis, protože výraz nebo funkce se může vytknout a přiřadit před daným voláním do nové proměnné.

Další omezení spočívá v zákazu vracet z jedné funkce více různých návratových typů na různých místech při daných typech argumentu. Tato podmínka je zavedena kvůli způsobu přetěžování funkcí v C++, kde se rozeznává podle argumentů a ne podle návratové hodnoty. Proto může být pouze jeden typ návratové hodnoty na danou definici funkce v C++.

Pokud si uživatel chce předat matici do vlastní definované funkce, musí si předávanou proměnnou pojmenovat stejně, jak se jmenuje parametr dané funkce a určit daný typ ještě před definicí této funkce. Poté může hodnotu uvnitř proměnné pozměnit, ale ne už typ, se kterým se již počítá při vyhodnocování funkce. Příklad tohoto použití je zobrazeno na obrázku 6.2.



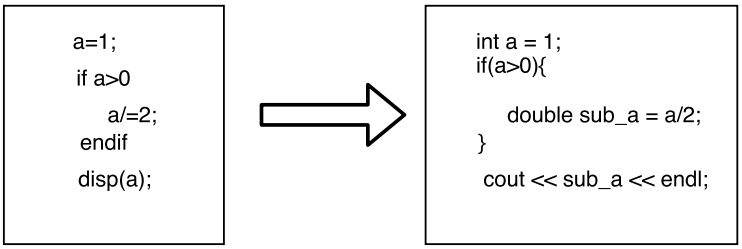
Obrázek 6.2: Struktura pro předáni matice do parametru funkce.

Implicitně jako typ argumentu je považován typ double. Na tuto skutečnost upozorňuje varovné hlášení při překladu. Pokud se předává skalární hodnota, kde nevadí převod na neceločíselné číslo, můžeme určit typ proměnné budoucího argumentu i po definici funkce.

Omezení předávání argumentů je způsobeno nutností znalostí typu jednotlivých parametrů při jejich zpracování ve funkci. Předávání matic v argumentech je limitováno způsobem předávání použitý v jazyku Octave a to předávání hodnotou. Více o předávání hodnot je zmíněno v manuálu [8].

Proměnná Množství přiřazení ve výrazu je limitováno překladačem na maximálně jedno. Výraz s vícenásobným přiřazením se vždy dá rozepsat do dvou čí více výrazů. Překladač neumožňuje použít v pojmenování proměnných jména vestavěných Octave konstant.

Důležitou podmínkou je brát na zřetel způsob redefinice proměnných. Způsob redefinice je způsoben překladačem, který vygeneruje novou proměnnou před danou proměnnou, která má být předefinovaná. Tím zaniká její platnost po opuštění bloku kódu. Daný typ proměnné je držen po celou funkci. Tím pádem při použití v jiném bloku kódu C++ překladač vypíše chybu. Řešením toho problému je zvolit správně typ, se kterým chceme pracovat. Například při práci s necelými čísly explicitním přidáním desetinné tečky v inicializaci proměnné. Příklad špatného použití je uveden na obrázku 6.3.



Obrázek 6.3: Problém se změnou typu proměnné.

6.4 Testování

K ověření správné funkcionality překladače jsou připraveny testy. Spouští se příkazem `make test`. Pokud jsou soubory kopírované z CD, je nutné nastavit práva na spuštění skriptu `test.sh` ve složce `tests`. Provádění testů trvá delší dobu v závislosti na hardwaru stroje (cca 60 sekund), kvůli překladu šablonového kódu překladačem C++. Výsledek testů je zobrazen v terminálu a také uložen v souboru test_results.out ve složce tests.

6.5 Důvody použití

Překladač je určen pro zjednodušení převodu kódu z jazyka Octave do systémů kompatibilních s jazykem C++. Nástroj má poskytnout automatizaci převodu.

Transpiler poskytuje převod základních programových struktur, 39 druhů operací, 3 datové typy, 4 vestavěné konstanty, podporu komentářů, 13 vestavěných Octave funkcí a možnost definovat uživatelem nové funkce.

Cílová platforma též snižuje režijní náklady (angl. overhead) tím, že nemusí interpretovat zdrojový kód, ale provádí pouze samotný výpočet.

Samotná výpočetní jádra Octave a knihovny Eigen jsou při různých výpočtech rozdílně výkonná[1], proto zde nejsou porovnány jejich výkonnosti.

Některé konstrukty jazyka Octave nelze přeložit na srovnatelně efektivní operace jazyka C++. (Jiné konstrukty pro efektivní zápis dané operace). Doporučuje se použít u překladu překladače přepínač "-O3", který odstraní nadbytečně rozepsané konstrukce generované transpilerem.

Překlad přináší úsporu paměti na cílové platformě. Na rozdíl od samotného interpretu Octave, který zabírá 21 MB paměti[2], binární soubor vyžaduje přibližně 500 KB–2 MB paměti. V porovnání s Matlab interpretem je poměr úspory ještě markantnější. Běžná instalace zabere 2.2 GB[3]. Pokud je potřeba ještě zmenšit cílový spustitelný soubor, je k dispozici přepínač překladače g++ paměťové optimalizace "-Os", který sníží nároky na paměť dokonce na 50–200 KB.

Informace o paměťové a časové náročnosti z celé podkapitoly jsou shrnuty v tabulce F.1 v příloze F, kde jsou zaznamenány délky jednotlivých běhů a velikost spustitelných souborů v překladu testovací sady s danými optimalizacemi.

[1] http://eigen.tuxfamily.org/index.php?title=Benchmark
[2] https://ftp.gnu.org/gnu/octave/
[3] https://www.mathworks.com/support/sysreq.html

Kapitola 7

Závěr

Vytvořený překladač je určen pro zjednodušení převodu kódu z jazyka Octave do systémů kompatibilních s jazykem C++. Nástroj poskytuje automatizaci převodu mezi jazyky. Díky tomu se urychlí nasazení vědeckých programů, psaných často v Octave nebo Matlabu, do produkce.

K sestavení překladače jsou využity poznatky z formálních jazyků v kapitole 2.1. Charakteristika a porovnání jednotlivých překládaných jazyků jsou rozebrány v kapitole 3, která dává představu o možných implementačních úskalích.

O kontrole správnosti fungování překladače se starají testy, které pokrývají jeho hlavní části. Díky nim je možné pokračovat v budoucím vývoji s možností kontroly správnosti nástroje.

Samotný transpiler poskytuje podporu pro převod základních programových struktur, 39 druhů operací, 3 datových typů, 4 vestavěných konstant, komentáře, 13 vestavěných Octave funkcí a uživatelem definované nové funkce. Dosavadní podporované překladové konstrukce jsou určeny tabulkou v příloze A.

Podařilo se vytvořit transpiler, který slouží pro automatizaci převodu kódu v Octave/Matlab do systémů podporující jazyk C++ (např. do frameworku OpenCV). Překlad neslouží pouze k zajištění kompatibility, ale též výrazně snižuje paměť potřebnou pro spuštění programu až o 99%.

V budoucnu je možné překladač rozšířit o další funkcionalitu. Konkrétně jde rozšířit o další možné datové typy (např. pole, sekvence, řetězce, komplexní čísla), nebo převodem dalších vestavěných funkcí jazyka Octave, které je snadné začlenit do překladače. Mezi tyto funkce můžeme řadit sort, generování náhodných čísel a statistické funkce.

Dalším rozšířením, které by zvýšilo uživatelský dojem z používání aplikace, by bylo grafické uživatelské rozhraní. Přínos pro uživatele by znamenal též integrovaný nástroj pro formátování zdrojového textu v C++, který by poskytl uživateli lepší orientaci v generovaném zdrojovém kódu, ulehčující mu upravit kód přímo v C++.

Pro zlepšení celkové rozšiřitelnosti překladače bych při další implementaci zvolil pro lexikální, syntaktické a částečně sémantické analýzy některý z nástrojů popsaných v podkapitole 2.4, kvůli lokalitě změn při přidání jednotlivých nových konstrukcí.

Velkou změnou by bylo koncipování překladače jako víceprůchodového (namísto současného jednoho), což usnadní odvozování typů proměnných ve volání funkcí. Tím pádem by se mohly dané typy ve funkcích vyhodnocovat až při volání dané funkce. Odlišný způsob implementace by způsobil zánik některých omezení překladače.

Překladač však má i určitá omezení. Tato omezení jsou převážně dána nutnými podmínkami překladače a rozsahem implementovaných jazykových konstrukcí. Podmínky při

použití jsou dány implementací překladače a určují způsob užívání jazyku Octave. Určený způsob užívání nijak neomezuje sílu daných podporovaných konstrukcí, pouze určuje, jak je má uživatel zapisovat, což může znesnadnit zápis a prodloužit výsledný program.

Literatura

[1] Aho, R. S., Alfred V.; ULLMAN, J. D.: *Compilers, principles, techniques, and tools*. Addison-Wesley Publishing Company, 1987, ISBN 0-201-10088-6.

[2] Beazley, D.: *PLY (Python Lex-Yacc)*. 2017, [Online; navštíveno 11.02.2017]. URL http://www.dabeaz.com/ply/

[3] Bornat, R.: *Understanding and Writing Compilers*. 2008. URL http://www.eis.mdx.ac.uk/staffpages/r_bornat/books/compiling.pdf

[4] van den Brand, M.; aj.: *Introduction to Generic Language Technology* . 2007, [Online; navštíveno 22.03.2017]. URL http://www.meta-environment.org/doc/books/getting-started/glt-intro/glt-intro.pdf

[5] van den Brand, M.; aj.: *Using The Meta-Environment for Maintenance and Renovation. IEEE*, 2007, ISSN 1534-5351, [Online; navštíveno 22.03.2017]. URL http://homepages.cwi.nl/~jurgenv/papers/CSMR-2007.pdf

[6] Eaton, J.; aj.: *Summary of important user-visible changes for version 3.8*. 2013, [Online; navštíveno 27.01.2017]. URL https://www.gnu.org/software/octave/NEWS-3.8.html

[7] Eaton, J.; aj.: *Free Your Numbers*. Free Software Foundation, Čtvrté vydání, Listopad 2016. URL https://www.gnu.org/software/octave/octave.pdf

[8] Eaton, J.; aj.: *Call by Value* . 2017, [Online; navštíveno 26.04.2017]. URL https://www.gnu.org/software/octave/doc/interpreter/Call-by-Value.html

[9] Guennebaud, G.: *Eigen a c++ linear algebra library*. Květen 2013, [Online; navštíveno 30.01.2017]. URL http://downloads.tuxfamily.org/eigen/eigen_aristote_may_2013.pdf

[10] Guennebaud, G.: *Eigen Main page*. Leden 2017, [Online; navštíveno 27.01.2017]. URL http://eigen.tuxfamily.org/index.php?title=Main_Page

[11] Johnson, M.: *LALR Parsing* . 2012, [Online; navštíveno 21.03.2017]. URL https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/140%20LALR%20Parsing.pdf

[12] Johnson, S. C.: *Yacc: Yet Another Compiler-Compiler*. 2006, [Online; navštíveno 11.02.2017].
URL http://dinosaur.compilertools.net/yacc/

[13] Klint, P.; aj.: *The Meta-Environment*. 2006, [Online; navštíveno 11.02.2017].
URL http://www.meta-environment.org/

[14] Krispel, J.; aj.: *Decaf JS*. 2017, [Online; navštíveno 21.03.2017].
URL https://github.com/rainforestapp/decaf

[15] Kulkarni, R.; aj.: *Transpiler and it's Advantages. International Journal of Computer Science and Information Technologies*, 2015, ISSN 0975-9646.

[16] Lesk, M. E.; Schmidt, E.: *Lex – A Lexical Analyzer Generator*. 2006, [Online; navštíveno 11.02.2017].
URL http://dinosaur.compilertools.net/lex/

[17] Meduna, A.: *Automata and Languages*. Springer-Verlag London, 2000, ISBN 1-85233-074-0.

[18] Meduna, A.: *Elements of compiler design*. Auerbach Publications, 2008, ISBN 1420063235.

[19] Meduna, A.: *Formal Languages and Computation*. Taylor & Francis Group, 2014, ISBN 978-1-4665-1345-7.

[20] Meduna, A.: *Formální jazyky a překladače*. FIT VUT Brno, 2015, [texty k přednáškám].

[21] Nikolaos, K.; aj.: *A basic linear algebra compiler for embedded processors*. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015, s. 1054–1059.

[22] Parr, T.: *ANTLR*. 2014, [Online; navštíveno 11.02.2017].
URL http://www.antlr.org/

[23] Quinlan, D.; aj.: *ROSE compiler infrastructure*. 2017, [Online; navštíveno 11.02.2017].
URL http://rosecompiler.org/?page_id=31#ProjectOverview

[24] Rahman, M. A.: *Eigen: A C++ Linear Algebra Template Library*. RWTH Aachen university, 2016, [Online; navštíveno 30.01.2017].
URL http://hpac.rwth-aachen.de/teaching/sem-accg-16/slides/07.Rahman-Eigen.pdf

[25] Rogel, S. J.: *ESSENTIAL MATLAB® and OCTAVE*. CRC Press, 2015, ISBN 978-1-4822-3464-0.

[26] Rouse, M.: *framework* . 2017, [Online; navštíveno 21.03.2017].
URL http://whatis.techtarget.com/definition/framework

[27] Waite, W. M.; Goos, G.: *Compiler Construction*. Únor 1996.
URL http://symbolaris.com/course/Compilers/waitegoos.pdf

[28] Češka, M.; aj.: *Překladače*. Vysoké učení technické v Brně: SNTL, 1986.

Přílohy

Příloha A

Překladová tabulka jazyků

Octave	Eigen	Popis
Komentáře		
`# komentář`	`// komentář`	jednořádkový komentář
`#{ komentář #};`	`/* komentář */`	víceřádkový komentář
Přiřazení		
`a=1;`	`double a = 1;`	přiřazení skaláru
`a=[1 2];`	`MatrixXd a(1,2);` `a « 1,2;`	přiřazení vektoru
`a=[1 2;3 4];`	`MatrixXd a(2,2);` `a « 1,2,3,4;`	přiřazení matice
Řídicí struktůry		

`if (a > b)`
`a += 1;`
`elseif (b > c)`
`b -= 1;`
`else`
`c *= 2;`
`endif`
	`if (a > b) {`
	`a += 1;`
	`} else if (b > c)`
	`{`
	`b -= 1;`
	`} else {`
	`c *= 2;}`
	`}`
		větvení if-then

`switch (a)`
`case {1, 2}`
`b += 2;`
`case 3`
`c *= 3;`
`otherwise`
`c /= 3;`
`endswitch`
	`/* switch (a) {`
	`case 1:`
	`case 2:`
	`b += 2;`
	`break;`
	`case 3:`
	`c *= 3;`
	`break;`
	`default:`
	`c /= 3;`
	`break; }`
		větvení switch

Octave	Eigen	Popis

`for i = [1 2];`
`i`
`endfor`
	`std::array<double, 2> a{1, 2};`
	`for (int c = 0; c < a.size(); c++)`
	`{`
	`std::cout << a[c] << std::endl;`
	`}`
		for cyklus

`while (a > 0)`
`a -= 2;`
`endwhile`
	`while (a > 0) {`
	`a -= 2;`
	`}`
		while cyklus

`do`
`a -= 2;`
`until(a < 0)`
	`do {`
	`a -= 2;`
	`} while(a >= 0);`
		do-while cyklus

`break;`	`break;`	přerušení cyklu
`continue;`	`continue;`	znovu vyhodnocení podmínky cyklu
Funkce		

`function rval = fun(a,b)`
`ret = a + b;`
`return;`
`endfunction`
	`double fun(double a, double b) {`
	`return a + b;`
	`}`
		definice funkce

`ret = fun(1,2);`	`double ret = fun(1,2);`	volání funkce
Zobrazení proměnné		
`a`	`std::cout << "a"= a << std::endl;`	výpis výsledku výrazu
`disp(a);`	`std::cout << a << std::endl;`	výpis proměnné
Aritmetické operace		
`++a;`	`++a;`	prefixový inkrement
`a++;`	`a++;`	sufixový inkrement
`--a;`	`--a;`	prefixový dekrement
`a--;`	`a--;`	sufixový dekrement
`a + b;`	`a + b;`	součet
`a - b;`	`a - b;`	odečítání
`a * b;`	`a * b;`	násobení
`a / b;`	`a * b.inverse();` `nebo a / b`	dělení
`a \ b;`	`a.inverse() * b;` `nebo b / a`	levé dělení
`a ^ b; nebo a ** b;`	`pow(a,b);`	umocnění
`a += b;`	`a += b;`	přičtení k proměnné a přiřazení
`a -= b;`	`a -= b;`	odečtení od proměnné a přiřazení
`a *= b;`	`a *= b;`	vynásobení proměnné a přiřazení

Octave	Eigen	Popis
`a /= b;`	`a *= b.inverse();` nebo `a /= b;`	vydělení proměnné a přiřazení
`a \= b;`	`a = a.inverse() * b;` nebo `a = b / a`	levé vydělení proměnné a přiřazení
`a ^= b;` nebo `a **= b;`	`a = pow(a,b);`	umocnění proměnné a přiřazení
Aritmetické operace nad každým prvkem		
`a .+ b;`	`a + b;`	součet
`a .- b;`	`a - b;`	odečítání
`a .* b;`	`a * b;`	násobení
`a ./ b;`	`a * b.inverse();` nebo `a / b`	dělení
`a .\ b;`	`a.inverse() * b;` nebo `b / a`	levé dělení
`a .^ b;` nebo `a .** b;`	`pow(a,b);`	umocnění
`a .+= b;`	`a += b;`	přičtení k proměnné a přiřazení
`a .-= b;`	`a -= b;`	odečtení od proměnné a přiřazení
`a .*= b;`	`a *= b;`	vynásobení proměnné a přiřazení
`a ./= b;`	`a *= b.inverse();` nebo `a /= b;`	vydělení proměnné a přiřazení
`a .\= b;`	`a = a.inverse() * b;` nebo `a = b / a`	levé vydělení proměnné a přiřazení
`a .^= b;` nebo `a .**= b;`	`a = pow(a,b);`	umocnění proměnné a přiřazení
Logaritmy a exponenty		
`log(a);`	`log(a);`	logaritmus
`exp(a);`	`exp(a);`	exponent
Logické operace		
`a & b;`	`a && b`	Konjunkce po prvcích
`a | b;`	`a || b`	Disjunkce po prvcích
`a && b;`	`a && b;`	Logická konjunkce
`a || b;`	`a || b;`	Logická disjunkce
`!a` nebo `not(a)`	`!a`	Negace
Konstanty		
`pi`	`M_PI`	Pí
`e`	`exp(1.0)`	Eulerovo číslo
`NaN`	`NAN`	Není to číslo
`Inf`	`INFINITY`	Nekonečno
Funkce nad maticemi		
`a'`	`a.transpose()`	Transponování
`det(a)`	`a.determinant()`	Diskriminant
`inv(a)`	`a.inverse()`	Inverze
Zaokrouhlovací funkce		
`ceil(a);`	`ceil(a);`	Zaokrouhlení na horu

Octave	Eigen	Popis
`floor(a);`	`floor(a);`	Zaokrouhlení dolů
`fix(a);`	`trunc(a);`	Zaokrouhlení oříznutím
`round(a);`	`round(a);`	Zaokrouhlení
Goniometrické funkce		
`sin(a);`	`sin(a);`	Sinus
`cos(a);`	`cos(a);`	Cosinus
`tan(a);`	`tan(a);`	Tangent

Příloha B

LL gramatika

LL-gramatika je definovaná čtveřicí G(N,T,P, <program>).
N={<program>, <blocks>, <block_n>, <block>, <exp>, <param_n>, <params>, <def_func>, <else-if>, <else>, <cases>, <case_choose>, <next_choose>, <otherwise>, <term>, <delimiter>, <int>, <return_value>}
T={function, if, switch, for, while, do, new_line, continue, break, otherwise, endfunction, endif, elseif, else, endswitch, (,), case, else, elseif, ",", ";", {, }, const_int, const_double, const_mat, id_int, id_double, id_mat, id_f, endwhile, +, -, return, EOF, #}

	Pravidlo r	Predict(r)
1	<program> → <blocks> EOF	function, if, switch, for, ;, while, do, new_line, continue, break, const_int, const_double, const_mat, id_int, id_double, id_mat, id_f, (, +, -, return, #
2	<program> → EOF	EOF
3	<blocks> → <block> <delimiter> <block_n>	function, if, switch, for, ;, while, do, new_line, continue, break, const_int, const_double, const_mat, id_int, id_double, id_mat, id_f, (, +, -, return, #
4	<block_n> → <blocks>	function, if, switch, for, ;, while, do, new_line, continue, break, const_int, const_double, const_mat, id_int, id_double, id_mat, id_f, (, +, -, return, #
5	<block_n> → ε	otherwise, endif, endswitch, endfunction, case, else, elseif, endfor, endwhile, until, EOF
6	<exp> → #	#
7	<block> → function <return_value> <def_func> <block_n> endfunction	function
8	<def_func> → id_f (<params>)	id_f

	Pravidlo r	Predict(r)
9	<params> → <id> <param_n>	id_int, id_double
10	<params> → ε	)
11	<param_n> → , <id> <param_n>	,
12	<param_n> → ε	)
13	<block> → if (<exp>) <block_n> <else-if> <else> endif	if
14	<else-if> → elseif (<exp>) <block_n> <else-if>	elseif
15	<else-if> → ε	else, endif
16	<else> → else <block_n>	else
17	<else> → ε	endif
18	<block> → switch (<int>) <may_del> <cases> <otherwise> endswitch	switch
19	<cases> → case <case_choose> <block_n> <cases>	case
20	<cases> → ε	otherwise, endswitch
21	<case_choose> → (<int>)	(
22	<case_choose> → { <int> <next_choose> }	{
23	<next_choose> → , <int>	,
24	<next_choose> → ε	}
25	<otherwise> → otherwise <block_n>	otherwise
26	<otherwise> → ε	endswitch
27	<block> → for <id> = <term> <delimiter> <block_n> endfor	for
28	<block> → while (<exp>) <block_n> endwhile	while
29	<block> → do <block_n> until (<exp>)	do
30	<block> → continue	continue
31	<block> → break	break
32	<block> → <exp>	#
33	<block> → ε	;, new_line
34	<term> → <const>	const_int, const_double, const_mat
35	<term> → <id>	id_int, id_double, id_mat
36	<delimiter> → new_line	new_line
37	<delimiter> → ;	;
38	<int> → const_int	const_int
39	<int> → id_int	id_int
40	<id> → id_int	id_int
41	<id> → id_double	id_double
42	<const> → const_int	const_int
43	<const> → const_double	const_double
44	<const> → const_mat	const_mat
45	<id> → id_mat	id_mat

	Pravidlo r	Predict(r)
46	<case_choose> → <int>	id_int, const_int
47	<may_del> → <delimiter>	new_line, ;
48	<may_del> → ε	case, otherwise, endswitch
49	<return_value> → <id> =	id_int, id_double, id_mat
50	<return_value> → ε	id_f
51	<block> → return	return

Příloha C

LL-tabulka

	function	otherwise	if	endif	switch	endswitch	for	...
<program>	1		1		1		1	
<blocks>	3		3		3		3	
<block_n>	4	5	4	5	4	5	4	
<block>	7		13		18		27	
<exp>								
<param_n>								
<params>								
<def_func>								
<else-if>				15				
<else>				17				
<cases>		20				20		
<case_choose>								
<next_choose>								
<otherwise>		25				26		
<term>								
<delimiter>								
<int>								
<id>								
<const>								
<may_del>		48				48		
<return_value>								

Tabulka C.1: První část LL tabulky

	;	while	do	new_line	continue	break	(	)	...
<program>	1	1	1	1	1	1	1		
<blocks>	3	3	3	3	3	3	3		
<block_n>	4	4	4	4	4	4	4		
<block>	33	28	29	33	30	31	32		
<exp>							6		
<param_n>								12	
<params>								10	
<def_func>									
<else-if>									
<else>									
<cases>									
<case_choose>							21		
<next_choose>									
<otherwise>									
<term>									
<delimiter>	37			36					
<int>									
<id>									
<const>									
<may_del>	47			47					
<return_value>									

Tabulka C.2: Druhá část LL tabulky

	case	else	elseif	,	{	}	endfor	const_int	...
<program>								1	
<blocks>								3	
<block_n>	5	5	5				5	4	
<block>								32	
<exp>								6	
<param_n>				11					
<params>									
<def_func>									
<else-if>		15	14						
<else>		16							
<cases>	19								
<case_choose>					22			46	
<next_choose>				23		24			
<otherwise>									
<term>								34	
<delimiter>									
<int>								38	
<id>									
<const>								42	
<may_del>	48								
<return_value>									

Tabulka C.3: Třetí část LL tabulky

	const_double	id_int	id_double	id_f	EOF	#	...
<program>	1	1	1	1	2	1	
<blocks>	3	3	3	3		3	
<block_n>	4	4	4	4	5	4	
<block>	32	32	32	32		32	
<exp>	6	6	6	6		6	
<param_n>							
<params>		9	9				
<def_func>				8			
<else-if>							
<else>							
<cases>							
<case_choose>		46					
<next_choose>							
<otherwise>							
<term>	34	35	35				
<delimiter>							
<int>		39					
<id>		40	41				
<const>	43						
<may_del>							
<return_value>		49	49	50			

Tabulka C.4: Čtvrtá část LL tabulky

first(#) = {const_int, const_double, const_mat, id_int, id_double, id_mat, id_f, - -, ++, -, +, (}

	const_mat	id_mat	endwhile	until	+	-	...
<program>	1	1			1	1	
<blocks>	3	3			3	3	
<block_n>	4	4	5	5	4	4	
<block>	32	32			32	32	
<exp>	6	6			6	6	
<param_n>							
<params>							
<def_func>							
<else-if>							
<else>							
<cases>							
<case_choose>							
<next_choose>							
<otherwise>							
<term>	34	35					
<delimiter>							
<int>							
<id>							
<const>	44	45					
<may_del>							
<return_value>		49					

Tabulka C.5: Pátá část LL tabulky

	return	endfunction
<program>	1	
<blocks>	3	
<block_n>	4	5
<block>	51	
<exp>		
<param_n>		
<params>		
<def_func>		
<else-if>		
<else>		
<cases>		
<case_choose>		
<next_choose>		
<otherwise>		
<term>		
<delimiter>		
<int>		
<id>		
<const>		
<may_del>		
<return_value>		

Tabulka C.6: Šestá část LL tabulky

Příloha D

Precedenční tabulka

	(	)	2	3	4	5	6	7	8	9	10	11	12	i	$
(	<	=	<	<	<	<	<	<	<	<	<	<	<	<	
)		>	>	>	>	>	>	>	>	>	>	>	>		>
2	<	>	<	>	>	>	>	>	>	>	>	>	>	<	>
3	<	>	<	>	>	>	>	>	>	>	>	>	>	<	>
4	<	>	<	<	>	>	>	>	>	>	>	>	>	<	>
5	<	>	<	<	<	>	>	>	>	>	>	>	>	<	>
6	<	>	<	<	<	<	>	>	>	>	>	>	>	<	>
7	<	>	<	<	<	<	<	>	>	>	>	>	>	<	>
8	<	>	<	<	<	<	<	<	>	>	>	>	>	<	>
9	<	>	<	<	<	<	<	<	<	>	>	>	>	<	>
10	<	>	<	<	<	<	<	<	<	<	>	>	>	<	>
11	<	>	<	<	<	<	<	<	<	<	<	>	>	<	>
12	<	>	<	<	<	<	<	<	<	<	<	<	<	<	>
i		>	>	>	>	>	>	>	>	>	>	>	>		>
$	<		<	<	<	<	<	<	<	<	<	<	<	<	

Tabulka D.1: Precedenční tabulka

Čísla v záhlaví a v prvním sloupci odpovídají operátorům jednotlivých řádků z tabulky 5.1

Příloha E

Precedenční gramatika

Pravidla pro precedenční analýzu:

Precedenční gramatika je definovaná čtveřicí G(N,T,P,E).

N={E, OMIT}

T={(,), i, ++, - -, ', **, ^, .**, .^, +, -, !, *, /, \, .\, .*, ./, <, <=, ==, >=, >, !=, & , |, ||, &&, =, +=, -=, *=, /=, \=/, .*=/, ./=, .\=, .^= }

	Přechod P
1	E → (E)
2	E → i
3	E → E++ || E- -
4	E → E'
5	E → E **/^/.**/.^ E
6	E → +E || -E || !E || ++E || - -E'
7	E → E */'/'/\/.\/.*/'./' E
8	E → +E || -E
9	E → E +/- E
10	E → E </<=/==/>=/>/!= E
11	E → E & E
12	E → E | E
13	E → E && E
14	E → E || E
15	E → E =/+=/-=/*=/'/='\=/ .*=/'./='/.\=/.^= E
16	OMIT → + || -

Příloha F

Experiment časových a paměťových nároků

Druh optimalizace	Bez optimalizace	Optimalizace -Os	Optimalizace -O3	Optimalizece -O3 a -Os
Spuštění Octave	10.302 s	9.889 s	12.991 s	9.984 s
Interpretování .m souborů	305 ms	444 ms	420 ms	332 ms
Překlad transpilerem	158 ms	142 ms	213 ms	178 ms
Překlad g++ překladačem	53.626 s	59.436 s	70.399 s	60.831 s
Spuštění binárních souborů	64 ms	68 ms	64 ms	63 ms
Potřebná paměť	2.044 MB	666,184 KB	680,496 KB	666,184 KB

Tabulka F.1: Tabulka zobrazuje rozdíly časové a paměťové náročnosti jednotlivých optimalizací.

Experiment je prováděn na jednom počítači nad testovací sadou (skládající se z 25 testů) přiloženou k programu. Výsledky mají orientační charakter – testy, pro každou kombinaci optimalizací, byly spouštěny čtyřikrát. Výsledky zaznamenané v tabulce jsou aritmetickým průměrem těchto měření.

Příloha G

Obsah přiloženého média